Plan Nets:

A Formal Representation of Action and Belief

for Automatic Planning Systems.

Mark Edwin Drummond

Ph.D.

University of Edinburgh

1986

EX BIBL. UNIV. EDINBURGEN.

For M & D

I believe that it would be worth trying to learn something about the world even if in trying to do so we should merely learn that we do not know much. This state of learned ignorance might be a help in many of our troubles. It might be well for all of us to remember that, while differing widely in the various little bits we know, in our infinite ignorance we are all equal.

– Karl R. Popper

Acknowledgements.

I take the opportunity offered by this space to say thanks to all the people and institutions who have made this work possible.

First, a special thanks to my parents, for encouraging me to make the trip to Scotland, and for providing help wherever and whenever it was needed. Thanks to Anja, for putting up with many instances of manifest unreasonableness and for always managing to come up with good suggestions. None of this work would have been possible without the multifaceted help of Mike Uschold. His determination to help, in spite of my determined intransigence, has not only made this work possible, but actually quite enjoyable.

I would like to thank Christian Weber, Eckhart Shultz, Martin Hammerle, and Jose Freitas for many interesting wine and pasta sessions, and for helping with the general education of a poor Canadian boy as to the workings of the world (Europe and Brazil, anyway).

Peter Ross, Graeme Ritchie, Ajay Gupta, Alex Lascarides, Ken Currie, Richard Tobin, Alan Black, Terry Patten, and Robert Stern have provided many helpful comments on this work as it has developed: to each, I extend my thanks. Peter and Graeme have been exceptionally patient in talking things over with someone who has no right to ask as many questions as I have. Karl Kempf and Peter Cheeseman have offered sympathetic ears and useful comments on a semi-annual basis, as permitted by conferences. I must thank Peter for introducing me to the devastating "So What?" question. I hope that this dissertation contains a satisfactory answer.

This work has been made possible only through the enthusiastic supervision and support of Austin Tate. It was Austin who got me into planning, and who over the last two years has answered all of my questions, great and small. Austin has also been instrumental in the construction of a fine working environment at the AI Applications Institute. The Institute and its people have helped this work in more ways than I can explain. As a first year supervisor, Robin Popplestone showed no shortage of constructive criticism and thesis ideas.

Financially, this work has been generously supported by the University of Edinburgh, through a Postgraduate Studentship; by the British Government, through an Overseas Research Student award; and by the British Council, through a Foreign and Commonwealth Office Scholarship. To all these institutions, I extend my thanks. The Students' Finance Board of Alberta also helped out by making me a loan for my first year of study at Edinburgh under very reasonable terms.

Generally, I'd like to thank all the people who work at the Department of AI, Department of Cognitive Science, and the AI Applications Institute for making Edinburgh such a fine place to spend a few years learning about AI, good beer, fine food, and theatre.

Abstract

Plan nets are a formal representation of action and belief for automatic planning systems. This thesis motivates, defines, demonstrates, and analyses plan nets. They are motivated through an argument regarding the inherent inability of current methods of action description to correctly describe the changes to an agent's beliefs under the occurrence of an action. Existing plan structures are reviewed in order to understand their strengths and weaknesses. Building on this understanding, plan nets are defined as action ordering constructs which provide a solid account of action teleology while permitting description of iteration, parallelism, and action disjunction. Mechanisms are defined for plan net projection which employ a definition of the causal independence of a set of actions. Unlike many other formalisms, plan nets distinguish between orderings on actions that are required by causality, and those that reflect the agent's preferences regarding overall plan outcome. Plan nets adequately characterise the relationship between actions and an agent's world model; because of this, plan net operators allow for a statement of both the "sensory" and "motor" results of an action. With current representations this is difficult or impossible. Plan nets are used to define the conditions under which a given world model assertion is believed, assumed, disbelieved, or not believed by an agent. Many examples of plan nets and their projections are given.

Declaration

I declare that this thesis has been composed by myself and that the work it describes is my own.

Table of Contents

1. Introduction and Overview. 1
1.1. Introduction. 2
1.2. The plan net desiderata. 2
1.3. Our approach and assumptions. 5
1.4. A plan of the thesis. 7
1.4.1. The basic thesis story. 7
1.4.2. Suggestions to the reader. 8

2. The Interaction of Action and Belief. 10
2.1. Chapter overview. 11
2.2. STRIPS-form operators: an example and problems. 12
2.3. Motivation: examples of desired behaviour. 21
2.3.1. Sensation. 21
2.3.2. External actions. 22
2.3.3. Inferences. 22
2.4. The new ontology. 23
2.5. The new epistemology. 27
2.6. What's been said about this already? 32
2.6.1. Moore: knowledge and action. 32
2.6.2. Konolige: knowledge, belief and action. 36
2.6.3. Doyle and de Kleer: reason maintenance systems. 37
2.6.4. Haas: a syntactic theory of belief and action. 38
2.7. Chapter conclusion. 40

3. Building Plans out of Operators. 43
3.1. Chapter overview. 44
3.2. The nature of a plan: type of structure and descriptive power. 45
3.2.1. State-space plan representations. 45
3.2.2. Action-ordering plan representations. 48
3.2.3. Summary. 52
3.3. Searching for plans. 54
3.3.1. The search space. 54
3.3.2. The linearity assumption. 59
3.3.3. Summary. 61
3.4. What have planners done in the past? 62
3.4.1. Green's system. 62
3.4.2. Kowalski's system. 63
3.4.3. STRIPS. 65
3.4.4. Hacker. 66
3.4.5. Interplan. 67
3.4.6. Waldinger's system. 68
3.4.7. Warplan. 69
3.4.8. NOAH. 70
3.4.9. Nonlin. 71
3.4.10. Deviser. 73
3.4.11. SIPE. 75
3.4.12. Molgen. 76
3.4.13. Georgeff's system. 77
3.4.14. O-Plan. 77

3.5. What do most current planners do? 78
3.5.1. Overview. 78
3.5.2. Common features: desirable and undesirable. 78
3.5.3. On analysing action-ordering plans. 82
3.5.4. Summary. 87
3.6. Chapter summary and conclusion. 88

4. Plan Net Definition. 90
4.1. Chapter overview. 91
4.2. Structure. 92
4.3. Consistency. 102
4.4. Projection. 108
4.5. Assumptions and reasons. 119
4.6. Defining problems. 121
4.7. Execution. 123
4.8. Summary. 133

5. Example Plan Nets. 135
5.1. Chapter overview. 136
5.2. Crossing the road. 137
5.3. Hammering a nail. 140
5.4. Telling Larry about the meeting. 143
5.5. Phoning Mike. 145
5.6. Stacking blocks. 148
5.7. Performing a litmus paper test. 152
5.8. Planning to infer mortality. 155
5.9. Finding out the colour of the block. 159
5.10. Planning to enable an external event. 162
5.11. Dialing a combination safe. 164
5.12. Checking the pilot light. 166

6. Discussion. 170
6.1. Chapter overview. 171
6.2. Comparing STRIPS operators and plan nets. 172
6.2.1. Basic operator representation. 172
6.2.2. Methods of operator application. 174
6.2.3. Plan net as triangle table. 175
6.3. Relation to Procedural nets. 178
6.4. Relation to Reason maintenance systems. 180
6.4.1. Basic ideas. 180
6.4.2. The basic correspondence. 182
6.4.3. Truth maintenance and net projection. 186
6.4.4. Representation: an example. 191
6.4.5. Some corresponding definitions. 191
6.4.6. Using multiple states. 199
6.4.7. Some general comments. 206
6.5. Teleological information (goal structure & plan rationale). 206
6.6. A simple STRIPS-like construction algorithm. 208

7. Suggestions for Future Research. 218
7.1. Chapter overview. 219
7.2. More execution advice. 219
7.3. Distinguishing between sensory and inferential actions. 219
7.4. System accountability. 220
7.5. Action hierarchy. 220
7.6. Use of a black-board architecture. 220
7.7. Retrojections. 221
7.8. Plan net construction. 222
7.9. Dependency directed backtracking. 222
7.10. Representing conditional proofs. 222
7.11. Which projection to use? 223
7.12. Possible search spaces. 223
7.13. Typed preconditions. 223
7.14. Plan execution monitoring. 224
7.15. The nature of iteration. 224
7.16. Keeping track of current reasons. 225
7.17. Importing more net theory ideas. 226

8. Conclusions. 227
8.1. Chapter overview. 228
8.2. Technical results. 228
8.3. Lessons learned. 230

References. 232

Appendix A: Prolog Implementation of Belief Consistency Maintenance. 238
Appendix B: Example Runs of Prolog Code. 248
Appendix C: The Order-Independence of a Detached Set of E-elements. 252

Chapter 1: Introduction and Overview.

I had toward the poetic art a quite peculiar relation which was only practical after I had cherished in my mind for a long time a subject which possessed me, a model which inspired me, a predecessor who attracted me, until at length, after I had molded it in silence for years, something resulted which might be regarded as a creation of my own; and finally, all at once, and almost instinctively, as if it had become ripe, I set it down on paper.

– Goethe

1.1. Introduction.

A thesis is a proposition; a theory submitted for discussion or proof. This thesis involves a new representation for plans, called *plan nets*. This dissertation presents the argument for, and definition of, plan nets, essentially by showing where current representations and ideas fall short of what is required.

This first chapter presents an introduction to the thesis and overview of the dissertation. It includes sections which deal with problem motivation, our research goals and approach, and the basic contributions of the thesis. A thesis story is given which lays out the structure of our argument. Some suggestions are given for getting through this dissertation, using the thesis story as a guide. This all starts in the next section, where the plan net desiderata are presented.

1.2. The plan net desiderata.

An agent plans when it considers in advance the actions that are to be employed in achieving its goals. The result of planning is a plan: a construct or mental artifact of the agent which describes the selected actions and their order of application. The class of all possible plans for an agent can be defined by giving the agent's plan syntax. In this dissertation we define a particular class of plans, and label each plan within this class a *plan net*.

What follows is a list of the basic features and abilities that we demand of plan nets. Many points are raised, and some arguments are started but not completed. Chapters 2 and 3 present a full discussion of the issues touched on by these introductory comments. While this section is intended primarily to motivate, it can also be taken as a statement of the results of the thesis presented in this dissertation.

Teleology

Information regarding *action teleology* must be included. The word "teleology" is used here to suggest interpretation in terms of *purpose*. Teleological information must be present to facilitate plan construction reasoning (through the detection of condition interactions), and

to enable effective plan execution monitoring (through analysis of the impacts of any given plan failure). Teleological information can be considered to map out the basic causal relationships among the actions specified by a plan.

Action-on-node structure

Plan nets must be action-on-node structures. This means that a plan net must be constructed out of action characterisations and ordering relations. Action-on-node structures are appealing due to their small space requirements and intuitive visual appeal.

Iteration

A plan net must be able to characterise simple iterative behaviours through its action-ordering structure. To include this ability might not appear difficult, but when considered in tandem with the action-on-node requirement iteration poses serious problems. Many action-on-node representations define only one ordering relation, interpreted as *before*. Such representations cannot describe iteration through the structure of a plan, since the *before* relation is irreflexive and transitive, thus asymmetric. No cycles means no repeated actions.

How action affects belief

Plan nets must correctly characterise the effects of an action on an agent's state of belief. It must be possible to describe both the *physical* and *mental* results of an action. Consider the difference between the results of "sensory actions" and "motor actions"; the former cause some beliefs for an agent, and the latter cause some new relationships or properties to hold in the agent's environment. Plan nets must be able to adequately characterise both sorts of results for an agent.

Parallelism

It must be possible to describe parallel actions. Such an ability is useful for two reasons. First, a plan construction algorithm can use a least-commitment action-ordering plan construction strategy. Second, realistic plans must take into account parallel actions, since

an agent will often be executing plans concurrently with the occurrence of actions in its environment.

Understanding causal independence

The representation must include a definition of the conditions under which actions are causally independent. An appropriate definition will form the basis of any least-commitment plan construction algorithm.

Harmony with reason maintenance

While we do not require that plan nets provide all the functions offered by a reason maintenance system, we do ask that they begin to tie together some of the ideas currently in the reason maintenance and automatic planning literature. In particular, using a plan net it must be possible to avoid some of the pitfalls involved with simple chronological backtracking.

Action disjunction

It must be possible to represent conditional actions within a plan net. Without such an ability, realistic plans and planning are impossible.

Simple plan analysis

A formal account of plan net analysis must be given. The analysis should be useful for performing "question-answering" (or *projection*) functions, and for doing general plan construction reasoning. We do not ask that the analysis given support *all* current methods of automatic plan generation, but it must be useful for some.

Agent intention vs. causal possibility

Certain sequencing constraints on actions are determined by causal possibility, and others are determined by the agent's intentions. Plan nets must responsibly distinguish one sort of sequencing constraint from the other. If this is done correctly, it might become possible to characterise planning as "intention management", instead of as simple action sequence

generation.

External actions

Plan nets must say something about the relationship between actions that are performable by a reasoning agent, and those that are "externally" activated. This is necessary if the agent is to reason about the effects of actions in general. Current representations do not say much about how this can be done.

1.3. Our approach and assumptions.

This research project adopts a certain methodology, and it is useful at the outset to make clear what this entails. We have the following methodological goals.

Parsimony

Plan nets should introduce as few notions as possible to achieve their stated aims. A parsimonious base with rich rules of composition is seen as more attractive than a bulging armoury of base-level constructs.

Coherence

There must be no inconsistencies in the plan net theory. Incompleteness is tolerable; indeed, unavoidable, but what *is* said must be coherent.

Generality

The plan net theory must be extremely general. The definitions must be domain independent, and open to varied application. Of course, we do not demand that plan nets solve all problems, but we do require a reasonable range of applicability for the solutions presented.

Intuitiveness

What is defined must closely follow intuition. Formalising general but incomprehensible

theories is of little use. People use only what they understand. We would like plan nets to be understood by as many people as possible.

Precision

Statements must be precise. To this end, we adopt some of the notions and notation of Net Theory (Reisig, 1985). Communication must *not* be purely in the form of documentation describing a working computer program. Programming language independent results are badly needed in the area of automatic planning.

Implementability

Technically infeasible constructs and analyses must not be called for. While we do not define the plan net theory in terms of a working computer program, we do require that what is said be simply and directly translatable into code. As a demonstration of this, we include as an appendix a Prolog implementation of a core component of the plan net framework.

We use the term *agent* throughout the dissertation and take this to mean a system of sensors and effectors, mediated by computer control. The sensors are to feed information regarding the environment into the memory of the computer, and the computer is to process this information and send commands to the effectors. On receipt of the appropriate commands, the effectors are to achieve some results in the environment.

This thesis makes no psychological claims. We discuss concepts such as *beliefs*, *goals*, and *plans* in abstract terms, and do not attempt to connect what is said with current psychological concepts or theories. This is a thesis in artificial intelligence, not psychology or, indeed, philosophy.

We are not attempting to address situations in which there are multiple agents. We define a representation for a *single* agent's plans, and do not concern ourselves with the provision of a theory suitable for reasoning about what other agents could come to "know" or "believe".

A point of "no distinction": we use the terms *action* and *event* interchangeably. In some circles, these terms are used to denote different notions of time and/or change. We use the terms as stylistically required.

And a last brief comment on our notational conventions. We use small capitals for acronyms, and write most other names as proper nouns. For example, STRIPS, GPS, Deviser, and O-Plan will all come up in the discussion at various points.

1.4. A plan of the thesis.

This section presents a thesis story, and uses the story to provide some suggestions for the reader. The thesis story takes the form of a chapter-by-chapter abstract of the dissertation.

1.4.1. The basic thesis story.

We start off in chapter 2 by setting out the basic principles of our theory of how actions can affect the beliefs of an individual agent. The basic problem addressed is that of correctly characterising the changes to an agent's beliefs under the occurrence of an action. We use some examples to show that classical ideas are inadequate, and suggest a new way of thinking about the relationship between action and belief. This chapter also reviews related work, and shows how it relates to what we are trying to do.

Chapter 3 discusses some issues of basic plan structure. Two primary kinds of plan structure are distinguished: action-ordering and state-space. We show that plans of either sort may or may not be able to describe parallel action execution. This understanding is used to explicate the relationship between the "linear assumption" and non-linear plans. We argue that much of the existing commentary on this relationship is quite confused. The terms settled on in these first two introductory sections are used in a planning system review. Most of the "classical" planners are covered. The planning system review forms the basis of an itemisation of features that are desirable to have in a plan representation, and also, to highlight those that are clearly undesirable.

Chapter 4 presents the formal plan net definitions and corresponding graphical conventions for plan nets. Some simple blocks world examples are given, but the full power of the representation is not demonstrated until chapter 5, where a number of more interesting examples are presented. All examples make use of the graphical conventions developed in chapter 4.

Chapter 6 covers various aspects of the plan net formalism in more depth. The relationships between plan nets and STRIPS operators, reason maintenance systems, and modern representations for teleological information are discussed. We briefly explore the problem of automatic plan net generation.

Various suggestions are made throughout the dissertation regarding areas in which the current work requires extension. Chapter 7 collects these suggestions together in one place, and presents some ideas about possible directions for pursuing future research.

Chapter 8 is our conclusion. We present an itemised summary of the technical contributions of the thesis, and give a list of the major lessons that have been learned along the way.

1.4.2. Suggestions to the reader.

Some of this material has already appeared in print. A now obsolete form of plan nets is described in Drummond (1985a). An IJCAI paper (Drummond, 1985b) describing plan nets similar to those defined in this dissertation is easily accessible, but the material presented here represents a substantial departure from this earlier work. A brief exposition of the current formalism appears in Drummond (1986). This last paper presents most of the technical material of chapter 4, but does not include all the motivation for the definitions provided by chapters 2 and 3; and as well, does not contain any of the analysis presented in chapter 6. The definitions of chapter 4 deviate only slightly from those of this last paper, following on suggestions made by the audiences at the paper's first and second presentations. The notion of *valuation* employed here is more general than that used in the 1986 workshop paper, and some technical problems with the old definition of e-element *concession* have been overcome. The definition we give here for *belief* is more general, and the definition of *disbelief* more restrictive, than the definitions that have appeared previously.

For the reader interested only in technical definitions, chapter 4 is the place to start. The examples of chapter 5 are comprehensible without a full understanding of the technical material, and in fact, may be useful to get a feeling for where the definitions of chapter 4 will lead. Chapters 2 and 3 represent the main motivation for the material we present, so if a reader is unconvinced of what chapter 4 is trying to do, the points of debate can be found in these first two chapters, outwith the formal presentation.

Chapter 6 will not make much sense until chapter 4 is understood, since at times it depends quite heavily on plan net notation. This is particularly true of the section where the relationship to reason maintenance systems is discussed. The sub-sections which discuss STRIPS triangle tables, teleology and and plan net generation are not as technical, and can probably be grasped without having read chapter 4.

While chapter 7 is the last of the dissertation proper, it might be useful to glance at it early on, to see what the problems are that we leave unsolved.

Chapter 2: The Interaction of Action and Belief.

If there were a verb meaning "to believe falsely," it would not have any significant first person, present indicative.

– Wittgenstein

There can never be any reason for rejecting one instinctive belief except that it clashes with others. It is of course *possible* that all or any of our beliefs may be mistaken, and therefore all ought to be held with at least some element of doubt. But we cannot have *reason* to reject a belief except on the ground of some other belief.

– Russell

2.1. Chapter overview.

This chapter motivates those features of the representation for plans to be developed in chapter 4 which pertain to the interaction of action and belief. This is done by examining existing representations, and exposing some of their problems. We show that current operator representations and methods of application are limited. The basic problem we address is that of correctly characterising the changes to an agent's beliefs under the occurrence of an action. Classical formulations do not adequately deal with this problem.

There are two main issues running through the commentary of this chapter. First is the so-called *frame problem* (Hayes, 1973); that is, the problem of finding adequate "laws of motion". A law of motion is a law, or rule, used to infer the properties of new states, given old. In this chapter, we determine that the classic STRIPS (Fikes & Nilsson, 1971) method of characterising actions through operators containing precondition-lists, delete-lists, and add-lists is limited, and propose a new method for dealing with the frame problem. Notice that comments made here on the STRIPS operator representation apply equally to the representations employed in more recent planning systems, such as NOAH (Sacerdoti, 1975a, 1975b), Nonlin (Tate, 1977), Deviser (Vere, 1981), and SIPE (Wilkins, 1983). These newer systems extend the current planning paradigm in various ways (as discussed in chapter 3), but all essentially still adhere to the STRIPS convention of precondition-, add-, and delete-list action characterisations.

The second issue running through our discussion involves the *closed-world assumption*. This is the assumption whereby if a statement cannot be shown to be true, then it is assumed to be false. Interesting, realistic, and practical planning systems cannot afford to make this assumption. If a statement cannot be shown to be true, simply assuming it to be false is inadequate. In reality, the statement may be true or false: the only way for an agent to find out which is by intentional perceptive action, geared towards causing a belief for the agent about the relevant condition described by the statement.

We begin in the next section by discussing STRIPS-form operator representations. It is shown that problems arise if we are to understand an agent's world model assertions as being *true* or

false. We argue that the truth conditions for an agent's world model assertions are unavailable to the agent. Instead, we offer the idea of classifying assertions as being believed or not-believed. STRIPS-form operators are discussed using this new terminology, and it is shown how these operators incorrectly characterise the changes to an agent's beliefs under the occurrence of an action. In section 2.3, the discussion changes tack slightly, and gives some examples of planning behaviour that systems based on STRIPS-form operator representations cannot describe. The examples are used to suggest an action ontology which can avoid the problems itemised in section 2.2. In section 2.4, the ontology is formulated, and we distinguish the "external" effects of an action from its "internal" effects, and introduce the notion of event executancy. Section 2.5 uses this simple ontology to derive the fundamentals of a representation for plans. These fundamentals essentially form the epistemology in which the definitions of chapter 4 are based. Section 2.6 discusses related research which is concerned with the interaction of action and belief. Section 2.7 reviews and concludes.

2.2. STRIPS-form operators: an example and problems.

Planning means plan construction, and classically, plan construction has meant assembling plans from operator schemas. Plans are constructed to solve problems, where problems are generally specified by a current world model and a set of goals. A reasonable way to begin then, is by examining the nature of an agent's world model, plans, and operator schemas.

An agent's *world model* is a database; a collection of assertions, perhaps in the predicate calculus, which describe the agent's environment. Each assertion is intended to determine some property or relation in the environment. For instance, the presence of the assertion *on(a,b)* in an agent's world model might mean that the object denoted by *a* stands in the *on* relation to the object denoted by *b*. If the agent lived in the simple (yet venerable) blocks world, this assertion might suggest that block *a* is supported by block *b*. Each assertion in the model is understood to be *true* or *false;* that is, to correspond with, or not correspond with (respectively), some feature of the agent's environment. (This correspondence notion of truth is discussed more fully below.)

A *plan* is a behavioural specification, or a characterisation of an action or actions. As such, any given plan will make reference to certain sorts of primitive events, or actions, in the world. The events that a plan can reference will be drawn from the ontology on which the plan representation language is based. Generally, current operator languages clearly distinguish only one sort of action. These are "motor" actions, performable by the agent to effect changes in its environment. There is one notable exception to this rule. Deviser (Vere, 1981) distinguishes timed external events from motor events. However, as Deviser's operator representation uses STRIPS-like techniques (discussed below), it still suffers from the problems itemised later in this section.

An *operator schema* is a *parameterised* characterisation of an action or actions. Thus, we might have an operator schema *move(X,Z,Y)*, which denotes the action of moving some object denoted by *X*, from an initial support denoted by *Z*, to a final support denoted by *Y*. In this context, *X*, *Y*, and *Z* are considered to be *variables*, and are *parameters* of the *move* operator schema.[1] An instantiation of such a schema is called an *operator*, and is obtained by substituting *constants* for the schema's variables. For instance, *move(a,b,c)* describes an action for moving the object denoted by *a* from the object denoted by *b* to the object denoted by *c*. Often, the distinction between operators and operator schemas is clear from context. In such situations, we use the word "operator" without confusion.

So operator schemas are action characterisations, to be used by an agent in constructing plans. Typically, operators describe an action in terms of the action's preconditions, postconditions, and name, following the operator representation conceived of in the STRIPS project (1971). Preconditions are understood to be those assertions which are required to be *true* for the action the operator describes to be enabled. The postconditions describe, through the mechanism

[1] Throughout this chapter, we will assume that variables are denoted by upper case letters, and constants by lower case letters, as done in the Prolog programming language.

of add-lists and delete-lists, the *effects* of the action. We might, for instance, notate the above-mentioned *move(X,Z,Y)* operator as follows.

Name: move(X,Z,Y).
Preconditions: on(X,Z), clear(X), clear(Y).
Postconditions.
Add: on(X,Y), clear(Z).
Delete: on(X,Z), clear(Y).

This specification is taken to mean that the object (denoted by) *X* can be moved from another object *Z* and placed on a third object *Y*, provided that initially, *X* is on *Z*, *X* is clear, and *Y* is clear. This formulation of *move* assumes that X, Y, and Z are bound to different constants, and that *clear(X)* means "*X* is totally clear", and that *on(X,Y)* means "Y uniquely supports X". After the action completes, *X* will be on *Y*, *Z* will be clear, *X* will no longer be on *Z*, and *Y* will no longer be clear. We will call any operator description similar to this one *STRIPS-form.*

The agent must use its world model in reasoning about the applicability of any given operator schema. The model provides the constants used to bind the variables of a schema to produce an operator. Through its add-lists and delete-lists, the derived operator will fully determine what assertions must appear and disappear in the world model when the operator is applied. Operator application is used to "simulate" action occurrence, so the additions to and deletions from the model are intended to be those that would be caused by the actual happening of the action being modelled.

For instance, if the assertions *on(a,b)*, *clear(c)*, and *clear(a)*, occur in the model, the agent may bind and execute an instance of the *move* schema: *move(a,b,c)*. According to the add-list of *move*, two assertions should be added to the agent's world model: *on(a,c)*, and *clear(b)*. The delete-list correspondingly specifies the removal of *on(a,b)*, and *clear(c)*. So in the application of this operator, two old assertions are deleted, and two new ones added.

There is a problem with this formulation of operators and their application, and it resides in the understanding that world model assertions must be either *true* or *false*. The essence of the problem is well addressed by Fodor (1981). In the following excerpt, Fodor discusses the status of a rule's (read operator's) compliance conditions (essentially, the operator's preconditions).

> Think of the rules as being like hypothetical imperatives; they have antecedents which specify conditions on mental representation, and they have consequents which specify what is to happen if the antecedents are satisfied. And now consider rules *a* and *b*.
>
> (a) If it's the case that *P*, do such and such.
>
> (b) If you believe it's the case that *P*, do such and such.
>
> Notice, to begin with, that the compliance conditions on these injunctions are quite different. In particular, in the case where P is *false but believed true*, compliance with *b* consists in doing such and such, whereas compliance with *a* consists in *not* doing it. But despite this difference in compliance conditions, there's something *very* peculiar (perhaps *pragmatically* peculiar, whatever precisely that may mean) about supposing that an organism might have different ways of going about attempting to comply with *a* and *b*. The peculiarity is patent in *c*:
>
> (c) Do such and such if it's the case that *P*, *whether or not* you believe it's the case that *P*.
>
> (Fodor, 1981, pp. 326-7)

Now in planning, we have a world model, which is essentially part of what Fodor has called an organism's (or agent's) "mental representation". In planning systems such as STRIPS, an assertion occurring in the model is assigned the semantic value *true*, or *false*. Under a *correspondence theory* of truth (Tarski, 1944), an assertion is true if, and only if, it corresponds to the facts. Thus we might say that the assertion *Snow is white* corresponds to the facts if, and only if, snow is, indeed, white. But as argued by Fodor, it seems highly dubious to claim that information about the correspondence of assertions with the facts could be available to an agent *qua* possessor of the model in which the assertions occur. Such information can only be formulated in a metalanguage; that is, in a language which can make statements *about* the relationship between object language assertions and their denotations. The impossibility of an agent's access to the truth conditions of an assertion is also argued by Moore and Hendrix (1982, pp. 124, 125).

So instead of saying that an assertion has two distinguished *truth* values, it might make more sense to say that each assertion is either *believed* or *not believed* by an agent. This can be done by distinguishing some subset of all possible assertions, and equating membership in this set with belief. We can call the set of all possible world model assertions M, and call the set of assertions currently in the world model B, such that $B \subseteq M$. If an assertion $p \in B$, for some agent A's belief set B, then we say that p is *believed* by A. If $\{p\} \cap B = \emptyset$, for agent A, then we say that p is *not believed* by A.[2]

[2] This story isn't quite what is formalised in chapter 4, but is accurate enough for the current discussion.

What basis is there for calling an agent's world model assertions "beliefs"? In line with Hayes (1981, p. 223), we consider that "by *belief* is meant any piece of information which is explicitly stored in the robot's memory.". It might turn out that this semi-technical notion of belief is inadequate for discussing the rich variety of human belief; it does, however, accord with our basic intuition of what a belief might be. Similar thoughts the nature of beliefs are expressed by Moore and Hendrix (1982). Levesque (1984, p. 198, footnote) says "Because what is represented in a knowledge base in typically not required to be *true*, to be consistent with most philosophers and computer scientists, we are calling the attitude involved here 'belief' rather than 'knowledge'.". An agent's "knowledge base" is its world model; we thus consider it fair to call assertions occurring in this world model *beliefs*.

Now the above account can be made simpler, since here we are not interested in formulating this idea of belief for a collection of agents, but instead, desire only an account suitable for a *single* agent. Thus, we can drop all references to any particular agent, and regard all beliefs as being held by our distinguished solitary agent. So if the assertion *Snow is white* occurs in the set of beliefs B, we say that our agent believes *Snow is white*. If *Snow is white* does not occur in B, we say that our agent does not believe *Snow is white*.

The philosophy adopted so far is quite similar to that used in the formulation of Doyle's (retroactively named) Reason Maintenance System (1979). We follow Doyle in arguing that "... we should study justified belief or reasoned argument, and ignore questions of truth. Truth enters into the study of extra-psychological rationality and into what common sense truisms we decide to supply to our programs, but truth does not enter into the narrowly psychological rationality by which our programs operate." (Doyle, 1979, p. 234). The full connection between Doyle's RMS work and the representation for plans developed here is discussed in chapter 6.

Where then does this leave us? Given that assertions are believed or not believed, it isn't hard to show that operators such as *move* no longer adequately characterise the changes to an agent's world model under the occurrence of an action. The problem is that STRIPS operators do not correctly describe what beliefs should come and go under action occurrence. It seems there's

no problem with operator *preconditions*: these are now thought of as beliefs which must be held prior to operator schema application. For instance, we can re-do the example first presented above, and reword it to fit the new interpretation as follows.

If the agent believes *on(a,b)*, *clear(c)*, and *clear(a)*, then it may bind and execute an instance of the *move* schema: *move(a,b,c)*.

The first indication of trouble brought on by the new interpretation appears when we examine the operator's *postcondition* specification. According to the add-list of *move*, two assertions should be added to the agent's world model: *on(a,c)*, and *clear(b)*. The delete-list specifies the removal of *on(a,b)*, and *clear(c)*. If deletions and additions are performed on the current set of beliefs, then two old beliefs are deleted, and two new ones added. But how much sense does it make to say that the performance of a *movement* in the world is the action which directly causes the agent to hold new beliefs about the physical results of the action? Of course, the agent might want to *assume* that the action did in fact accomplish what the operator specified; that is, that the movement did successfully transfer block *a* from *b* to *c*. And the world model assertion would then be some sort of *assumption* about the physical results of the block movement action. But it seems a mistake to suggest that the block movement action itself is responsible for adding the relevant formula to the agent's world model.

The technique of using add-lists and delete-lists to specify how an action changes the world model is key in the STRIPS solution to the frame problem. We offer a new solution to this problem in two parts. We discuss specific difficulties with the STRIPS addlist mechanism directly below. In section 2.5, the basic functionality of a STRIPS deletelist is addressed, and an alternative mechanism is presented.

There are two ways it seems reasonable to suggest that formulas can be *added* to generate successor world models. First, the agent can *assume* an assertion, and thus simply add the assertion to generate the successor state, without justification. We can consider STRIPS operator application to be doing this. Second, the agent may have the assertion added by some sensory action; that is, by some action which forms a sensory embedding of the agent in its environment.

For the block stacking example given above, a vision system might be employed to produce the belief *on(a,c)*. The belief would be asserted and justified by the vision system, and by the way the vision system "connects" the agent to its environment. The justified belief can only *indirectly* arise from the original stacking action, insofar as it was this stacking that caused *a* to be on *c*, and thus permitted the sensory system to generate *on(a,c)*; or insofar as the application of the move action gives the agent sufficient cause to assume *on(a,c)*.

This issue of the origin of belief about the outcome of motor actions is also discussed by philosophical action theorists. For instance, in attempting to explicate human volition, Davis (1979, p. 16), writes "A volition is an event which is normally a cause of the agent's belief that he is acting in certain way, *and* which normally causes such doing-related events as to make it true that he *is* acting in that way." Brand (1984, p. 40) has criticised this view: "It is odd to think that a single mental event causes *both* the associated behavior and the belief that the associated behavior is being brought about. The belief that the associated behavior is occurring, it would appear, results from perceptual stimuli, including afferent feedback. These perceptual stimuli occur after the bodily movement has begun.". Notice that while Davis and Brand are concerned with issues not directly addressed here (like volition), some common interests exist; in particular, the question as to what sort of action we can attribute the generation of belief. Our answer is similar to Brand's: beliefs can be generated only by assumption, or by a sensational action -- an action which connects the agent (human or artifact) to its environment.

The problem with using STRIPS-form addlists to characterise the *effects* of actions has to do with the classic distinction between knowledge and belief. We would say an agent *knows p* if that agent *believes p*, and further more, *p* is *true*. STRIPS-form operators are based more on the idea of knowledge, by not admitting that there may be discrepancies between the stated effects of an action and the actual effects of that action. Since STRIPS-form operators are (at least implicitly) based on this idea of knowledge, they conflate what the addlist formulae *denote* (describe, characterise) with how they are *used* by the agent. This much seems clear from the term "addlist". Each formula in a STRIPS-form operator addlist is intended to denote (describe, characterise) the actual physical results of the denoted action in the agent's environment.

However, if we say an addlist formula is *true* after operator application, then we fail to distinguish between the actual result of the action, and what the agent might *believe* to be the result of the action.

There is another way to look at this problem. Consider using STRIPS-form operators to plan sensory, information gathering actions. Now it is clear that since the agent's world model is distinctly finite in capacity, and that since the world in which the agent exists is (presumably) infinite, there will be an infinity of statements *true* of the world, but of which the agent is completely unaware. However since STRIPS-form addlists say that something will be made to be the case *and* that it will also be believed by the agent to be the case, these operators cannot be used to plan information gathering actions. The essence of an information gathering action is this: some fact is *already* "true" of the world, but the agent has no beliefs regarding the fact. To gain a belief about the fact, an action must be invoked which can impart to the agent some relevant information. With STRIPS-form operators, an assertion can only be *true* after action application, but if it was true before, and the agent simply unaware that it was true, then a STRIPS-form operator cannot be used to characterise the action by which the agent *is made aware* of the truth of the assertion.

Consider the following example. An agent performs a technically perfect block movement operation: block *a* is moved from block *b* to block *c*. *This action* unarguably results in the statement "a is on c" being *true*, by simple virtue of the action's technical perfection. Suppose that, *the same movement* imparts to the agent a measure of block *a*'s weight. This could be easily accomplished through the use of a weight sensor in the agent's robot arm. So what are the *results* of this *single* block movement action? That *a* is on *c*, certainly, but also, that the agent has been given a "belief" about the weight of block *a*. These are *both* legitimate results of the movement action, and it is necessary for an operator description language to be able to discuss each.

To continue the example, consider what the agent's beliefs might be following the performance of the block movement action. The agent will believe that the weight of block *a* is a given value, certainly, since the weight sensor in the arm is connected to the agent's world model

so as to produce beliefs about the weight of transported objects. But does the agent believe that block *a* is on block *c*? On what basis would the agent believe this assertion? First, it could, on the basis of the operator's addlist specification, *assume* that the move worked, and generate a belief about *a* being on *c*. This belief would be an *assumption* about the action's outcome. Alternatively, the agent might assume nothing, and wait for the appropriate sensory input to generate a belief about *a* being on *c*. This belief would not be an assumption, since it would have a solid sensory justification.

So if we consider world model assertions to be beliefs, it seems we require *two* sorts of addlists for our operators: a list which describes the external, physical results of an action, and a list which describes the internal, mental results of an action. Henceforth, we will refer to these "addlists" as "result sets", since the term "result" does not commit us as to how the formulae will be used to generate successor world models, and since the term "set" is more technically accurate than "list".

In summary, the "add-list" argument is as follows. If we say that an operator has an effect which is to make an assertion *true*, then there is no immediate problem. Here, truth is a *relation* between a fact and a belief, and if the operator makes something *true* it deals with both the fact and the belief in a single step. If however, we say that an agent has no access to the truth conditions of its beliefs, then an operator must contain a specification of how the action it denotes affects both the *fact* and the agent's *belief* about the fact. Without truth to bind beliefs to facts and make these beliefs knowledge, we have *two* independent entities, and are required to say how any given action affects *both*. So operators must have two "result sets", to be used in operator application in a variety of ways.

The next section presents some examples of desired planning behaviour, so as to add concreteness to the above argument. The argument and examples are used in section 2.4 to formulate a simple but adequate ontology of action. Beginning with an idea of the sorts of action we wish to discuss will make it easier to understand this ontology.

2.3. Motivation: examples of desired behaviour.

Classically, planning has been defined as the business of stringing together action specifications which, when realised by an agent, will change the world in such a manner as to solve a stated problem. In these examples, we will refer to such actions as *motor* actions. Defining planning purely in terms of motor actions is restrictive. It's not hard to come up with reasonable example plans which must make reference to other *sorts* of action. In particular, the need often arises to make reference to *sensory* actions, *inferential* actions, and *external* actions. The word "reference" is used to suggest that plans do not *contain* actions, but rather, contain specifications of, or references to, actions. If a plan is to contain such references, it must be based on an ontology which makes the required distinctions. As suggested, the main distinctions to be drawn here are between sorts of events; namely, sensory, inferential, external, and motor. This 4-sort action division is motivated in the following paragraphs by three examples. (One example per new action type; it seems unnecessary to motivate the use of motor actions.)

2.3.1. Sensation.

The scenario for our first example is this: Alfred the arbitrary agent and I are in the coffee lounge whiling away our research hours. This problem I pose to Alfred: "There's a cardboard box in the hallway outside this room which contains a wooden block of a certain colour. Can you tell me what the colour of the block is?" If we ignore the problem of Alfred having to reason about how I could come to know something he knows, we can rephrase this problem as a slightly simpler one: Alfred must come to know the colour of the wooden block to which I have referred. It's not hard to imagine how he might reason. Perhaps something like this: "I need to know; how can I know the colour of a block? Well, by *seeing* it. How can I see it? Perhaps by being in close proximity to it, pointing my eyes at it, having my eyes open, and ensuring that the hall lights are on while doing so. OK, how can I be in close proximity to the block? By going to it... .". This sample problem should serve to demonstrate that planning can be construed as more than the stringing together of motor action specifications. During the course of plan construction, Alfred

made explicit reference to an action with a possible *sensory* result. We will refer to actions which have sensory results as *sensory actions*. Alfred understood the sensory action of *seeing* to have certain preconditions, and planned to satisfy these preconditions, in a way that seems compatible with existing *backward chaining* planning techniques. In view of this, planning could be thought of as the stringing together of sensory *and* motor action specifications. Once again, it's not hard to come up with examples which demonstrate that this slightly expanded characterisation of planning activity remains inadequate.

2.3.2. External actions.

A plan must be able to reference actions completely external to the agent constructing the plan. For example, imagine an entirely law-abiding agent who wishes to cross the road: before doing so, she must wait for the "walk" light, and ensure that the road is free of traffic. Some plan will be formed to control this behaviour. The plan must contain a reference to the "walk/don't-walk" pedestrian control mechanism, however crude, else the agent will have little reason to stand at the curb, awaiting the "walk" light. From this example, it seems reasonable to argue that plans should be able to contain references to actions in the agent's environment. We refer to actions that do not directly affect the agent's beliefs and which the agent cannot control as *external actions*.

2.3.3. Inferences.

Agents can plan to infer. To motivate this, one final example. Agent Alex wishes to conclude that Fred is Mortal, and she has only one rule of inference to do the trick. The rule states that if Fred is human, she can conclude that he is mortal. Lacking information about Fred's human-ness (not unreasonable, in a world populated with agents of arbitrary type), she might plan to carry out some tests to determine what sort of individual Fred is. Her final plan might contain some motor and sensory actions with various possible outcomes, and the rule of inference would be joined in (somehow) with the results of one of the sensory actions. Once the rule's antecedent (that Fred is human) is satisfied by Alex's beliefs, an application of it sanctions her further belief in Fred's

status as a mortal. We refer to an action which adds to the agent's beliefs, and over which the agent does have control, as an *inferential action.*

This concludes our brief series of examples. These examples are used in the next section to formulate a simple but adequate ontology in which the event types required by the examples can be described. We return to these examples in chapter 5, where plan nets are presented for each.

2.4. The new ontology.

With the examples of the last section in mind, a new definition of planning seems appropriate: planning can be thought of as the problem of generating a behavioural specification in advance of the behaviour's production. The general sorts of behaviours we want to consider have been presented in the examples above.

What we require is an ontology, that is, a way of carving up the world, which puts us in a position to discuss the sorts of actions required by the examples. A picture of such an ontology is given in figure 2.1. The picture is intended to distinguish two features of any given event and the results it achieves. We distinguish two parameters: where the effects of an action are achieved, which we refer to as the *locality of change* parameter; and which of the agent or environment executes the event, which we call the *event executancy* parameter.

It turns out that this ontology gives us a surprisingly rich vocabulary in which to discuss actions and their effects. Below, we motivate the ontology; in the following section, we present the basics of how this ontology could be represented to an agent.

The basic structure of the picture in figure 2.1 is common in AI and cognitive science. Two areas of interest in the picture can be distinguished: that part of the universe which is inside the agent, and that part which is outside. The inside component is often called "mental", and the outside component "physical". From a planning perspective, this is pretty well the simplest possible picture that one can draw: inside, we have the agent's world model, operators, goals, plans, etc., and outside, we have the real wooden blocks, rooms, and traffic lights which make up the agent's environment.

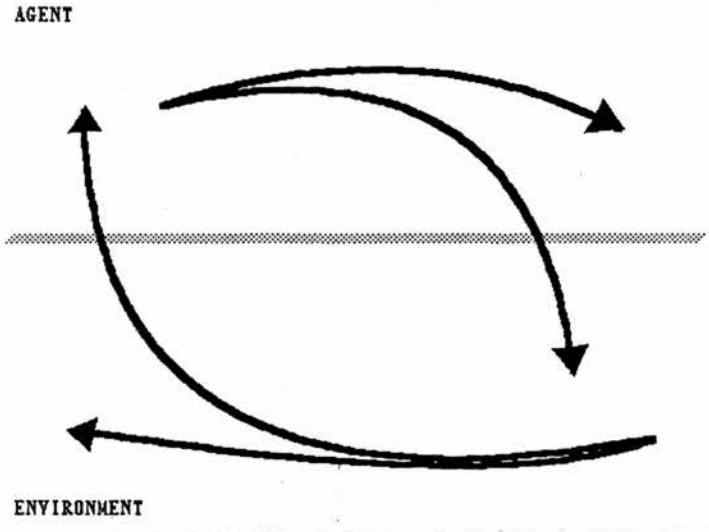


Figure 2.1: The basic ontology.

Given that there are two areas of interest, it makes sense to be precise about in which area an action achieves its result. We suppose that any given action can have results in either of the two locations, or *both*. We call the results of an action that are realised inside the agent *internal results*; and we call the results of an action that are realised outside the agent *external results*. *Locality of change* refers to where any given effect is realised by an action. In figure 2.1, this is shown by the location of the head of an arrow: an arrow-head inside the agent indicates an internal result, and an arrow-head outside indicates an external result.

In figure 2.1, we have clearly distinguished the agent from its environment. It makes sense then, to use this same distinction when explaining *event executancy*. We use this term to refer to the "activation source" for an action; that is, to refer to which of the agent *or* environment has *control* over whether or not the action will occur. This is an obvious distinction to make, and

follows naturally from the structure of the picture we have drawn. There will be some actions for which the agent has executancy, and some actions for which the environment has executancy. Notice that due to the lack of resolution in the environment part of the picture, we cannot (indeed do not want to) consider whether *another agent* is responsible for activating an event. At the level of abstraction considered here, all things outside the agent of concern are on par, and cannot be distinguished. Thus we will not distinguish between intentional action on the part of other agents and random physical processes, such as floods, fire, and earthquakes (assuming the agent lives in California).

The question is now one of utility: given the distinctions of event executancy and locality of change, what sorts of actions can we describe? Think back to the examples. Is it possible to derive the basic actions they seem to require?

It was suggested that agents can plan to *infer*. In terms of the ontology's distinctions, we could call an action *inferential* if for that action, the agent has executancy, and the action has *only* internal results. For instance, the action of the agent concluding q on the basis of p and $p \rightarrow q$ is inferential, since the agent has executancy, and the only result of the action is that the agent comes to believe q. This seems to correspond quite well with the standard notion of inference: an inference is something performed by an agent in order to generate more beliefs about its environment.

However, it seems that an action could also be called *sensory* if it has any internal results, independent of executancy. What might be the difference between a sensory and an inferential action? Given the distinctions made by the ontology, we cannot distinguish between them. The basic difference would seem to derive from the "source" of information for the beliefs generated by an action. A sensory action will use the situation actually pertaining in the environment to generate beliefs. In contrast, an inferential action would appear to take only information within the agent's world model into account. Sensory actions seem to "bring in" new information, while inferential actions simply "re-arrange" old information.

So we cannot distinguish between sensory and inferential actions, given our basic ontological distinctions. This isn't a serious problem, however. We cannot distinguish the two sorts of actions, but we can *discuss* both. In terms of our abstract labels, the two event types cannot be distinguished; but the labels we provide do allow us to discuss all relevant features of the two sorts of events. In particular, the effect of the actions on the state of belief of the agent is adequately determined by the locality of change parameter. So both sensations and inferences can be discussed, if not distinguished.

In the long run, it might be a good idea to remain non-committal about the difference between sensory and inferential activities, and how any given agent is connected to its environment by them. Depending on the exact nature of an agent's modalities, different things will be directly "sensible". The nature of the sensory resources available to any given agent will determine what that agent can sense, and what that agent must infer. We will return to this issue of the distinction between sensory and inferential actions at various points: in chapter 4, the interpretation of the defined formalism is explained; and in chapter 5, some examples are given, with accompanying discussion.

For an example of a sensory action, it seems reasonable to suggest that there might be an action of the agent *seeing* an "on" relationship between two blocks. This action is comprised of the functioning of the agent's sensory machinery: light strikes the agent's visual apparatus, low-level visual processing proceeds, signal interpretation is carried out, and *ex miro motu*, the assertion *on(a,b)* is added to the agent's world model. This is an *action* in any useful sense of the word. As stated by Hayes (1981, p. 229), "Observations are analogous to events."

An action could be called *motor* if the agent has executancy and the action has *any* external results. Thus, the action of the agent moving block *a* from *b* to *c* is motor: it causes *a* to be on *c*. If the action also causes the agent to believe something about the weight of block *a*, we might be willing to call it *sensory* as well.

An action might be classed as *external* if the environment has executancy, and that action has *only* external results. Trees falling over in forests, bridges collapsing, and lunar eclipses are

all examples of external events. These actions can occur independently of the agent, and do not affect the agent's beliefs. If however, an action has any internal results, even if the environment has executancy, then the beliefs of the agent may also be affected. But if the action has no internal results, then the agent will remain ignorant of the event's occurrence.

We will often not be afforded a clean separation between inferential, motor, sensory, and external actions. Witness the difficulties experienced when attempting to distinguish between sensory and inferential actions. Witness also the block movement action discussed above, which is both "motor" and "sensory". But practically, it doesn't really matter. These terms are only labels suggested by the examples. Provided that the formalism one adopts makes distinctions of locality of change and event executancy, discussion can use these underlying ontological terms, instead of the event-type labels suggested by the examples. The utility of the ontology only becomes truly apparent when the examples of chapter 5 are presented.

To sum up, this section has argued that it is reasonable to analyse actions in terms of two parameters: event executancy, and locality of change. Using these parameters, we can discuss inferential, motor, sensory, and external actions, at least to the degree of precision required by the examples of the last section.

In the next section, we return to the issue of how to characterise actions by operators. The ontology of this section is used to induce an epistemology that will be used as the basis for the definition of our operator description language in chapter 4.

2.5. The new epistemology.

In this section we define the basic epistemology of the formal definitions set forth in chapter 4. We build on the criticisms of section 2.2, the examples of section 2.3, and the simple ontology of section 2.4. The key issue is this: we are required to produce the fundamentals of a bit of syntax which describes to an agent the ontology of the last section.

The representation we propose still employs operator schemas to characterise actions for an agent. Each operator will have preconditions that determine when the operator is applicable. As

suggested by Doyle's (1979) RMS work, it is important to be able to describe enablement of *action* in terms of belief *or* lack of belief. In STRIPS, one could phrase a negative enabling condition using a *not* construction. However, since we are no longer operating with *truth*, some other sort of negative enabling precondition must be provided. We suggest that each operator specify two enabling sets of assertions: one set to describe those beliefs that *must* be held if the action denoted by the operator is to be enabled, and one set to describe those beliefs that *must not* be held. This arrangement allows operators to be enabled in situations where the agent has no beliefs regarding the negative enabling conditions.

Each operator will have two "result sets". The first, which we call the *internal results postset* of the operator, describes the results of the denoted action that will be realised *inside* the agent. The second we call the *external results postset*, and it describes the results of the denoted action that will be realised *outside* the agent. An operator's internal results postset therefore describes the beliefs that the agent can expect to hold upon completion of the denoted action, and the external results postset describes the physical results of the action in the agent's environment.

For example, the *move(a,b,c)* operator could be formulated so as to have the following postsets.

Internal results postset: { weight(a,X) }
External results postset: { on(a,c) }

This can be understood by the agent to mean that following execution of the action denoted by *move(a,b,c)*, a belief of the form *weight(a,X)* will be held, and a physical result of the object denoted by *a* being on top of the object denoted by *c* will be obtained. The variable *X* is a place-holder for some constant yet to be generated by the arm load-measurement device.

It must be possible for each operator to be used in two different ways. Given some operator *e*, and a "state" of belief *c*, we would like two different methods available for operator application. The first method of application we term *realistic*, and it calls for adding *only* the operator's internal results postset to generate a successor to *c*. This would seem to accurately model the action denoted by the operator: after the action's occurrence, the beliefs specified by the operator's internal results postset will be held, and the external results of the action will be

realised, but the agent will have no beliefs regarding these external results.

In order to simplify reasoning about operator application, we might expect the agent to *assume* that the external effects of the action have in fact been achieved. To do this, the agent could use the *union* of the operator's internal and external results postsets to generate a successor state to *c*. We call this second method of operator application *assumptive*. If this sort of operator application is to be useful however, we must provide a definition of *assumption*, such that given any belief state, we can discern legitimate, justified beliefs from assumptions (generated by various means, one of which is assumptive operator application). In chapter 4, we define these two operator application methods, and define what an assumption is, so as to be able to distinguish assumptions from justified beliefs.

And this actually brings us back to the issue of deletelists, so far quite carefully ignored. What beliefs "go away" upon operator application? We can no longer simply attach a "delete set" to each operator, since there are to be *two* methods of operator application. If an operator is applied in a realistic fashion, then only its internal results postset is added, and only those existing beliefs superseded by those being added should be removed. If an operator is applied assumptively, then both internal and external results postsets are added, and all beliefs that are superseded by the union of these sets should be nulled.

In seeking inspiration for a solution to this problem, we can consider how an agent actually does "remove" old beliefs, during its day to day existence. The agent exists continuously in an environment, connected to a plethora of sensory devices. These devices define the agent's various sensory modalities. Now as things change in the environment, various of the agent's beliefs will become outdated, and must be revised. We assume that the only *reason* the agent will have to reject an existing belief is that it can be shown to be incompatible with a belief generated by the incoming data stream.

Imagine a simple blocks world situation in which an agent believes *on(a,b)*, and is then made to believe *clear(b)* through the occurrence of a sensory action. How is the agent to revise its beliefs? Clearly, the block denoted by *a* cannot be on the block denoted by *b* since this latter

block is clear, and thus supports no other blocks. We might say that in the simple blocks world, the two beliefs of a block both being clear and supporting some other block are "inconsistent". To remain in possession of a consistent set of beliefs, the agent must somehow recognise inconsistencies, and delete those beliefs that can restore consistency.

The information required to maintain consistency has actually long been present in planning systems, and used to detect inconsistent states of affairs. For example, Warren (1974) used information about domain incompatibilities in his WarPlan system to detect when certain sequences of actions were impossible. Such information was used by Sacerdoti (1975a, 1975b) in the NOAH system to linearise unordered actions. Allen and Koomen (1983) used similar information when applying Allen's TimeLogic system (1981) to the planning problem. Lansky (1985) also uses the term "domain constraint". Following Allen and Koomen (1983), we use the term *domain constraint* to refer to that information available to an agent which describes an incompatibility within a particular domain state.

It is possible to view our domain constraints in terms of database management concepts. Some database systems make use of *integrity constraints*. These are general restrictions on allowable values for fields within database records. The sorts of constraints considered in database theory are much more general than considered here (see, for instance, Gray, 1984, pp. 265-259). By only considering one sort of relatively restricted constraint, we gain the advantage of being able to guarantee constraint satisfaction. With expressive constraint languages, it may be difficult (nigh impossible) to provide such a guarantee.

We suggest that an agent use this domain constraint information as follows. Using a set of domain constraints, it is possible to define when any given set of beliefs is consistent. In chapter 4, we employ such a definition of consistency to characterise a function, called the belief consistency maintenance function, which is used to suggest ways of restoring consistency. This function is defined with respect to a set of domain constraints, and is passed two sets of world model formulae: those that currently exist in the world model, and those that are to be added. The function returns all possible sets of "reconciliation" sets, such that the removal of any *one* of

these sets from the current world model makes the union of the current formulae and added formulae consistent with respect to the domain constraints.

We propose to use this belief consistency maintenance function in two ways. An implementation of the function will be used to suggest revisions as new information is made available to the agent through its various sensory modalities. It will also be used by the operator application mechanism when deciding what assertions to delete from one belief state in order to derive another. When applying an operator realistically, the function is passed the enabling set of assertions, and the internal results postset of the operator being applied. When applying an operator assumptively, the function is passed the set of enabling assertions and the *union* of the internal and external results postsets. In either case, those beliefs that are deleted are only a function of what is currently believed, and what is being added to the set of beliefs.

One can trace the seeds of the idea of using domain constraints back as far as 1973, to the paper in which Pat Hayes proposed using a set of "general laws" that each state must obey. While the basic idea was articulated early on, no planning system seems to have made use of it. The technique of using domain constraints to remove "outdated" beliefs offers, as suspected by Hayes, a unique and effective means of dealing with the frame problem.

The implied use of domain constraints can even be found in current texts on AI. See, for example, Nilsson (1980), pages 152 and 153. The idea as employed in this thesis is that instead of deleting beliefs directly through the application of operators, one deletes "old" beliefs only when they conflict with "new" beliefs. Exactly how this works is made clear in chapter 4, where we present an algorithm for maintaining consistency in a set of beliefs with respect to a set of domain constraints.

Locality of change was not the only parameter distinguished in section 2.4. The other part of the ontology involved what we call *event executancy*. From the perspective of our agent, event executancy is a question of *ability* to realise the action described by any given operator. If the agent *can* realise the action an operator describes, then the agent does have executancy over that action. If the agent cannot realise the action an operator describes, the the agent does not have

executancy over that action.

We can make this idea of action realisation more precise. It is possible to consider each operator as a procedure applied to specific arguments. The arguments are the constants used to derive the operator from the more general operator schema. At plan execution time, the "name" of the operator can be viewed as the address of an executable procedure, and the constants in the operator viewed as the procedure's arguments. If a mapping exists between the operator and one of the agent's primitive executable procedures, then the agent has executancy. If no such mapping exists, then the environment can be said to have executancy.

2.6. What's been said about this already?

This section discusses those research efforts which are concerned with theories of, and formalisms for reasoning about, knowledge and/or belief and action. Chapter 3 covers related research in the area of basic plan representation.

2.6.1. Moore: knowledge and action.

Moore (1985) describes a theory of knowledge and action that is based on a general understanding of the relationship between the two. His theory was originally presented in a doctoral thesis (Moore, 1980), and a short report of the same work also exists (Moore, 1977). His theory develops the pioneering work of Hintikka (1962) on the logic of knowledge and belief, and recasts McCarthy's situation calculus (McCarthy, 1968; McCarthy & Hayes, 1969) so as to mesh with Hintikka's possible worlds characterisation of knowledge.

On the surface, it might seem as though Moore's theory of knowledge and action should relate quite strongly to the work presented here. However, it is easy to show that the topics addressed by Moore's work are of a very different nature to the ones we address. In particular, Moore is concerned with providing a formal theory for reasoning about what agents can *possibly* know, whereas we are providing a representation for plans suitable for installation in an agent which uses the representation to build plans designed to solve specified problems. Certain of the

agent's problems may involve the agent in coming to believe new facts about its environment; if so, the representation must provide for the description of information gathering actions. However, when solving its problems, the agent will presumably operate in a goal-directed fashion -- the reasoning undertaken by the agent will pertain to the selection of operators relevant to solving the specified problem. Moore presents a theory of reasoning *about* what an agent could know, and does not touch on the representations that the agent might employ in actually coming to "know" any particular fact. This distinction is simple and clear, and yet easy to miss. The difference between the research efforts hinges on *reasoning*: Moore has presented a theory of reasoning *about* what arbitrary agents can *possibly* come to know -- we are concerned with providing a plan representation that any one of these agents can employ while actually reasoning about coming to "know" something. See Barnden (1986) for a short essay on the importance of the distinction; the paper also provides a reasonable source for references into the literature.

That Moore is concerned with reasoning about what agents could possibly come to know is evidenced by the fact that a possible worlds characterisation of knowledge commits his theory to assuming logical omniscience on the part of the agents being reasoned about. His theory assumes agents will actually draw all possible conclusions from what they are initially given. This is perhaps fine as a theory of reasoning about what agents *might* come to know, but is clearly inadequate as a theory of reasoning to be installed in one of the agents.

A possible worlds analysis also means that Moore's theory deals only with knowledge, and ignores questions of belief. Appelt (1985) has used Moore's theory as a basis for the reasoning component of a language planning system, and has made the following observation:

> One may argue that an adequate theory of language planning must be based on a theory of *belief* rather than a theory of knowledge. Although this is a valid point, an adequate theory of belief is difficult to formalize, because once one admits the possibility of holding beliefs that are not true of the world, a theory of belief revision and truth maintenance is required.
>
> (Appelt, 1985, p. 8).

Similar comments were made even earlier by Hayes (1973). In the following excerpt, Hayes is discussing the two sorts of beliefs a robot (what we've been calling an agent) must have regarding time and change:

> (a) There must be beliefs *about* time. For example, beliefs about causality. (b) The robot lives *in* time: the world changes about him. His beliefs must accommodate in a rational way to this change. ... The first is solely concerned with thinking: the second involves observation.
>
> (Hayes, 1973, p. 223).

If one attempts to adapt Moore's theory to deal with Reason Maintenance problems, serious difficulties quickly arise. It seems difficult or impossible to add in the required notions of *assumption*, *justification/reason*, and *premise*. In addition, the representation for plans given by Moore suffers from many of the problems to be discussed in chapter 3 (notably, lack of an ability to represent least commitment with respect to action-ordering, and a lack of any teleological information).

Perhaps most seriously, Moore's representation for action and change runs afoul of the frame problem. Since he uses McCarthy's situation calculus formalisation (McCarthy, 1968; McCarthy & Hayes, 1969), a frame axiom must be provided for each formula unaffected by an action. The number of frame axioms required in any non-trivial system is phenomenal. It was for just this reason that the STRIPS operator representation and method of application was seen as a clever move: the STRIPS assumption allowed a person writing action descriptions to ignore all the frame axioms required by the situation calculus approach, and permitted them to proceed with specifying their particular problem domain.

Our view, in which an agent has access only to its beliefs, can be contrasted with Moore's (1985) view, in which a distinction is made between the *physical* and *informational* prerequisites of a plan or operator: "... when an agent entertains a plan for achieving some goal, he must consider not only whether the physical prerequisites of the plan have been satisfied, but also whether he has all the information necessary to carry out the plan." (Moore, 1985, p. 319). This distinction does *seem* to make sense. Certain things must hold in the world for an action to be enabled, and certain information must be *known* to the agent if the plan is to be executable. For example, in order to open a key lock, the agent must be physically in control of a key; that is, the agent must be holding the key of the safe to be opened. It seems reasonable to contrast this with the situation that arises in the case of a combination lock. Here, the agent must *know* the combination to the lock, otherwise no plan formulated (save blowing the safe) will work. (This

example comes originally from McCarthy, 1977.)

While we might *want* to make the distinction between physical and informational preconditions, we certainly don't *need* to. So far, we have argued that all the agent has access to are its beliefs about the world. How then, would an agent's beliefs change to reflect the two lock scenarios just described? In the first one, there is a key lock, and the agent must have the required key. The agent could believe *opens-key-lock(key237,lock200)*, and *holding(me,key237)*. These beliefs might be those required to enable the *key-lock* operator. In the second case, we have a combination lock, and the agent must "know" the required combination. The agent might have a belief of the form *opens-combo-lock("45-25-17", lock500)*, and this could enable the *combo-lock* operator. In both examples, the agent must hold the required beliefs before attempting to execute the relevant action. If the agent is willing to attempt actions, while not believing that the action's preconditions hold, then either it is attempting an experiment (a test), or behaving irrationally.

What then, is the difference between a physical and an informational prerequisite? Well, certainly, if the beliefs of the agent don't correspond to the way the world really is, then the actions described by the operators stand little chance of success. For instance, the *physical* prerequisite of the key actually opening the lock, expressed in the agent's beliefs as *opens-key-lock(key237,lock200)*, must actually obtain, or the real action of opening the lock will not succeed. But we can turn this around and argue that, in just the same way, the *information* prerequisite of *opens-combo-lock("45-25-17", lock500)* must actually obtain in the environment, or the act of dialing the lock to open it will fail. That is, the combination lock must actually be physically configured such that dialing a "45" to-the-right, followed by a "25" to-the-left, followed by a "17" to-the-right does, indeed, result in the release of the hasp. This seems no different to saying that the key denoted by *key237* must actually be cut so as to turn the tumblers of the lock correctly. So while one might *like* to make a distinction between physical prerequisites and informational prerequisites, such a distinction isn't necessary.

2.6.2. Konolige: knowledge, belief and action.

Konolige (1980) presents a formalism which uses a syntactic approach for representing and reasoning about knowledge, belief, and action. An agent's beliefs are identified with formulas in a first-order language, called the object language (OL). Propositional attitudes such as knowing and wanting are modelled as a relation between an agent and a formula in the OL. Another language, the metalanguage, or ML, is used to study the OL. Using this ML/OL structure, Konolige is able to describe an agent's beliefs as a set of formulas in the OL (a theory), and express partial knowledge of that set of formulas.

Much of the content of Konolige's formalism is drawn from Moore's (1980) work on a possible worlds formalisation of knowledge and action. The main contribution of Konolige's work is to show that a syntactic approach, when coupled with a situation calculus description of action and change, can adequately formalise Moore's criteria for the interaction of knowledge and action.

The syntactic approach doesn't suffer from the problem of assuming logical omniscience on the part of the agents under scrutiny. This is because it is possible to describe explicitly in the ML the inference procedure any given agent might use. However, there are a number of problems left unaddressed by Konolige's work. In particular, Konolige's formalism continues to use a situation calculus approach to describing action and change. As mentioned previously, STRIPS was seen as an advance over the situation calculus, since when using STRIPS, one is not required to provide any frame axioms for each action description. STRIPS *assumes* that, if an assertion is not deleted by an operator application, the assertion is unaffected by the operator. This is now known as the *STRIPS assumption*. By using a situation calculus approach, Konolige is making a move back towards tedious and computationally intractable action specifications.

Konolige *explicitly* ignores the problem of belief consistency maintenance. He does acknowledge that this is a problem. In his formalism, once a belief is held about a situation, it is held for all time. Since beliefs can easily be incorrect, it is often necessary to revise them in the light of new information. In Konolige's formalism, if conflicting information is added to an

agent's theory (set of beliefs) then the theory simply becomes inconsistent. Furthermore, action descriptions do not include a specification of how they affect an agent's state of belief. Konolige asserts that there is no obvious and well motivated way to make modifications to axioms describing actions (and their related frame axioms) so as to take into account the agent's belief about a situation rather than what actually holds in the situation. He says that what is required is "a principled way of deriving the changes to an agent's beliefs that result from an event, given a description of the event as a relation on situations." (Konolige, 1980, p. 27).

Konolige (1980) presents an interesting example problem, in which an agent must reason about how to determine whether or not a fully enclosed (and thus unobservable) pilot light on a natural gas stove is burning. The solution is for the agent to attempt to get a burner on the stove to light by turning on the gas. If the burner ignites, the pilot light can be assumed to be on. If it does not ignite, the pilot light must be off. We present a plan of action which is a solution to this problem in chapter 5.

2.6.3. Doyle and de Kleer: reason maintenance systems.

As should be clear from much of the preceding discussion, the most closely related work which deals with belief is that of Doyle (1979). The relationship between the representation for plans we propose and Doyle's Reason Maintenance System (or RMS) is best explained with the definitions of chapter 4 and examples of chapter 5 in hand. For this reason, we postpone detailed discussion of RMS until chapter 6. There are however, a couple of distinctions between the stated goals of RMS and the representation for plans developed in this dissertation. The major points of distinction worth keeping in mind while reading chapter 4 are as follows.

An RMS package is viewed as a subsystem, and thus the dependency records it manipulates are strictly internal, essentially unavailable for inspection by any program utilising the RMS's service. This means that RMS structures are normally explained in terms of abstract *nodes*, since the RMS need attach no significance to any particular structure. It is the responsibility of the calling program to associate assertions (or whatever) with RMS nodes. However the organisation employed here is much simpler, and more rigid. We define all plan net structures directly in terms

of assertions (world model formulae), since the plan manipulation routines are expected to be an integral part of the overall planning system. We are not attempting to provide a general RMS service; rather, we are defining a representation for plans which enjoys many of the benefits realised by conventional RMS dependency structures.

We are dealing with modelling *action* and *change*, while RMS only exists to cope with model *changes*. We present a representation for *plans*, and must therefore integrate all the key ideas regarding plan representation generated over the past 20 years. Just what these ideas are is made clear in the next chapter. To quote Doyle (1979) from the original TMS (Truth Maintenance System) paper: "... much of the frame problem (e.g. how to give the 'laws of motion' and how to retrieve them efficiently) lies outside the scope of this discussion." (Doyle, 1979, p.235). These are exactly the problems we address.

One can view de Kleer's (1984, 1986a,b,c) Assumption-based Truth Maintenance System (ATMS) as an engineering-like enhancement of the basic idea. The ATMS does not include any notions of action or change either, and for our purposes, offers little more than the basic ideas put forth originally by Doyle. Chapter 6 covers the ATMS and its relationship to our work in more detail.

2.6.4. Haas: a syntactic theory of belief and action.

Haas (1986) presents an elegant syntactic theory of belief and action which improves on Moore's in three ways: 1) it does not predict that agents instantly believe everything that can be proved from their beliefs; 2) it gives a better account of what one must know about an object in order to know what that object is; 3) it gives a better account of when one needs knowledge to perform an action.

Moore postulates that an agent knows what an object X is if the agent knows that X is T, where T is a term that denotes the same object in all worlds compatible with the agent's beliefs. Haas shows that this proposal fails, since one can construct arbitrarily complex expressions that denote the same object in all possible worlds. Very complex expressions would appear to require

simplification to a form suitable for the task at hand. Haas suggests that knowing what something is depends on what it is to be used for.

Haas' theory is based on a subtle LISP-like system of quotation and evaluation, designed to get around the problems involved with quantifying into the scope of the predicates "know" and "believe". The system can be embedded within first order predicate calculus, and used to reason about what agents could come to believe, given some base beliefs from which to reason. In this thesis we are not primarily concerned with reasoning about what agents could come to believe by reasoning from a set of base beliefs. Rather, we are interested in a characterisation of how an agent's beliefs are changed by the occurrence of an action. In a planning context, this becomes the problem of devising an operator description language which correctly characterises actions and their effects. One must also give the means by which operators are to be applied; that is, rules must be given for translating one model into another under the application of an operator. Haas does not address this problem. He argues (as does Moore) that the representation of procedural knowledge is an important but separate problem. The work presented in this dissertation concentrates more on procedural knowledge than declarative knowledge.

Because Haas is not addressing the representation of plans he need not deal with many problems which we cannot avoid: teleology, conditional constructs, parallelism, and the causal independence of actions. Since we present a theory of plan representation, we must also address some issues that Haas need not; in particular, we must tell some sort of story regarding the application of individual operators, and the projection (forward simulation) of a plan from given initial conditions. These are key issues to be addressed in any theory of plan representation.

Haas has a simple yet adequate (for his purposes) theory of time: a set of instants totally ordered by $<$. It is not clear whether some more planning oriented view of time such as the situation calculus could be easily integrated with Haas' system. It would be quite interesting to try this.

Key to Haas' theory is the view of inferences as mental actions that the agent can plan to perform. This is exactly one of the suggestions made in this chapter. Agents have goals, some of

which can be satisfied by inferential actions; thus, these actions must be planned, since they will not simply happen instantaneously as soon as their antecedents are satisfied.

Haas equates recall in constant time with belief. Essentially, an assertion is believed by the agent if it is in the agent's world model: the retrieval of the assertion must not involve deduction, or we would not be prepared to say that the agent believes it. Although this is consistent with the introductory comments of this chapter, the story we formalise in chapter four is slightly different. There we claim that the presence of an assertion in the model is not a sufficient condition for it to be believed by the agent. It is believed only if it is in the model and has a *reason* to be in. If no reason for being in can be given, then the assertion is an assumption, not a belief. We do not equate not being in the world model with lack of belief; an assertion is disbelieved if it is not in the model and a reason can be given for this. If no reason can be given for the assertion not being in, then the assertion is simply not believed. We have not made this clear up until now since it would have served no purpose. This chapter has tried to show that current operator schemes incorrectly characterise the changes to an agent's world model under the occurrence of an action. For the argument it was adequate to equate membership in the world model with belief, essentially as done by Haas. In chapter 4, we will be more precise about the conditions under which an assertion is actually a belief of the agent.

2.7. Chapter conclusion.

To plan, an agent requires operator schemas, a world model, and goals. Recall that the essence of the planning problem is assembling operator schemas into plans on demand. This assembly is performed in terms of the agent's world model and its goals. We have seen that operator schemas are characterisations of actions from the perspective of the agent that uses them in plan construction.

Classically, the assertions that occur in an agent's world model have been classified as *true* or *false*. In this chapter we have argued that an agent cannot have access to this sort of correspondence information, and have suggested that world model assertions are best understood as being believed or not-believed by an agent.

The operator description languages used in all current planning systems derive from the STRIPS system (Fikes & Nilsson, 1971) idea of precondition-lists, add-lists, and delete-lists. We have argued that a simple "add-list" mechanism does not adequately describe the changes to an agent's beliefs under the occurrence of an action. If added formulas are not *truths*, but *beliefs*, then the source of belief creation must be responsibly distinguished. Actions which achieve results in the world do not themselves cause belief on the part of the agent regarding these results. It is easy to show however, that some actions *do* actually cause new beliefs to be held by an agent. An example action of block movement was given: this action generates a belief for the agent about the moved block's weight, and generates an actual condition of the moved block's being somewhere. The single movement action cannot be said to generate the belief about the block's new location. The agent may *assume* something regarding the action's physical result, but then this act of *assumption* is a different action from the *move*, and *it* is responsible for adding a formula to the agent's world model.

This is not terminological hair-splitting. It is easy to see that STRIPS-form operators cannot be used to characterise *sensory* actions; that is, actions by which the agent is made aware of features of its environment.

To motivate a solution to this problem, we gave some examples of behaviour it seems desirable to be able to describe, and used these example behaviours to form a new action ontology. The example behaviours seemed to depend of 4 "types" of action: motor, sensory, inferential, and external.

In section 2.4, an ontology was developed which makes enough distinctions to discuss these action types. The ontology distinguishes event executancy and locality of change. Event executancy refers to which of the agent or environment can execute the event, and locality of change refers to where the event achieves its results, either in the agent or in the environment.

In section 2.5 the ontology was used to induce the fundamentals of an epistemology in which our operator descriptions will be phrased. The basic idea is that the operators we define will have two sorts of enabling conditions and two sorts of effects. The enabling conditions are

either positive or negative, and describe beliefs that must or must not be held if the operator is to be applicable. The two sorts of effects are the *internal* and *external* results of the denoted action, as suggested by our ontology. Delete-lists were dispensed with, and a mechanism suggested for "removing" beliefs which depends only on how an operator is to be applied.

A mechanism for implementing event executancy was given which is based on the idea of a mapping from operator descriptions to "computer programs" realising the descriptions. For any given operator, if a mapping exists, then the agent has executancy, since an executable description of the action exists. If there is no such mapping, then the agent does not have executancy, since there is no way for the agent to realise the action.

This chapter has attempted to motivate those features of the representation to be developed in chapter 4 which pertain to the interaction of action and belief. However, nothing has been said yet regarding how one should link a collection of action descriptions together so as to form a coherent plan. This is the topic of the next chapter.

Chapter 3: Building Plans out of Operators.

...[A] purely sequential model does not truly reflect the real *causal structure* of processes. In any sequentializing view we can not differentiate whether two events occur one after the other because the first is a prerequisite of the second or whether this order in time is solely by chance. But, in fact, the causal relations are those which, to a large extent, characterize a system.

– Wolfgang Reisig (1985, p.2)

Attempt to end, and never stand to doubt; nothing's so hard but search will find it out.

– Robert Herrick

3.1. Chapter overview.

This chapter motivates those features of the representation for plans to be developed in chapter 4 which pertain to basic plan structure and expressiveness. We discuss issues of plan representation and construction and use the discussion to settle on some terminology. The terminology is employed in a planning system review, which is used in turn to describe the plan representations employed by most current planners. We argue that current representations are inadequate in many respects. The plan representation developed in this work builds on the criticisms and suggestions of this chapter.

We begin in section 3.2 by distinguishing two sorts of representation: state-space and action-ordering. It is shown that plans of either sort may or may not be able to describe parallel action execution. Comments are made which allow us to define the terms *parallel plan* and *serial plan.*

In section 3.3, the discussion turns to issues of plan construction. We distinguish two possible search spaces for a planning system: world states, and partial plans. It is shown that a planner's search space determines the sort of plan construction reasoning that it must do. These comments on plan construction are used to argue that parallel plans (some say "Nonlinear plans") do not address the "linear assumption".

The terms defined in the first two sections are used in section 3.4 to do a planning system review. Each major planner is briefly described.

In section 3.5, we use the comments of the review to determine the sort of plan representation used by the majority of current systems. Problems with current representations are discussed, and used to motivate the representation to be defined in chapter 4.

In what follows, we return to the standard practice of referring to assertions as true and false, in order to make the exposition simpler. Note however, that the terminology of belief and non-belief developed in chapter 2 could be substituted in, with no loss of explanatory value. However, since we're trying to explain how *actions* can be fitted together to make plans, we lose nothing by adopting the standard convention of referring to assertions as being "true" or "false".

Indeed, since our comments are often about previous planning systems and their representations, we would inaccurately characterise these systems if we used belief/non-belief terminology.

3.2. The nature of a plan: type of structure and descriptive power.

We begin by examining possible plan representations. Two main issues are of concern. First, there is the issue of whether the representation is state-space or action-ordering, and second, whether or not the representation allows for the parallel execution of planned actions. The AI community consensus seems to be that these issues are one; that is, that state-space plan structures cannot describe parallel execution, and that action-ordering plan structures are necessarily able to describe parallel activity. But this isn't true. The issues are genuinely orthogonal: some state-space representations can describe parallel activity, as will be argued below, and demonstrated in chapter 4; also, certain action-ordering plans are unable to represent parallel action execution.

3.2.1 State-space plan representations.

State-space problem representations form the backbone of much work in AI. A state-space representation is composed of two sorts of objects:

> ...*states*, which are data structures giving 'snapshots' of the condition of the problem at each stage of its solution, and *operators*, which are means for transforming the problem from one state into another.
>
> (Barr and Feigenbaum, 1981, page 32).

AI is not alone in employing state-space representations. As noted by Simon:

> The state-space representation is borrowed from the classical representations of physics and other domains of applied mathematics. In these domains, a set of basis variables is selected (position and velocity in the case of classical dynamics), and each space-time point is characterized by a vector of the values of these variables. The laws of the system, typically in the form of differential equations, are the 'move operators.'
>
> (Simon, 1983, p.15).

State-space representations are used in a number of diverse domains; for instance, in applied mathematics, economics, automata theory, and in providing operational semantics for programming languages. For a readable review of the use of state-space representations in various fields, see Simon (1983). In AI in general, and planning in particular, state-space representations are heavily

used. If we employ a state-space representation *for a plan*, then the "vectors" (to use Simon's term) in the representation will describe states of the world (or environment that the plan is to be executed in), and the move operators will describe the actions that are to be carried out in order to execute the plan.

Why should we want to use a state-space representation for a plan in preference to any other sort? The reason is that it is exceedingly easy to *recognise* a state-space solution to a given problem. Any state-space can be conveniently internally represented as a directed graph structure, where nodes are states, and arcs are operators which transform one state into another. Represented this way, a solution to a problem is a path through the structure which starts at one of the initial nodes, and terminates at one of the nodes satisfying the goal criteria. It is not hard to encode the search for a solution as a simple computer program which attempts to connect any two given nodes in the state-space data structure.

Of course, to say that a solution is easy to recognise is not to say that it is easy to generate. Far from it. The number of options open to a planning system at each stage of plan development is often huge. But this doesn't alter the fact that a state-space representation makes the *recognition* of a plan easy.

Part of the simplicity of reasoning provided by state-space representations comes from the way states "package up" related information. If, while reasoning forward from an initial state, a planner wishes to link an operator into the plan, all that's required is a check to see that the operator's preconditions are satisfied. The preconditions are satisfied if a state can be found in which the preconditions already exist, or perhaps, from which the preconditions can be derived (say by proof). Applying the operator involves only binding its variables to constants found in the selected state. After binding, the operator can be used to derive a new state, (say) according to its add-list and delete-list specifications. If the preconditions of the operator hold in the state in which it is applied, then the new state derived is guaranteed to be "reachable" from the old by the laws of physics the operator encodes. If the planner is to reason backwards, we can "turn around" the application process, and unapply operators in a backward manner, working from what

is required to what currently exists.

Forward or backward, the analysis of a state-space plan structure is simple. It is this simplicity of analysis which makes the state-space representation so convenient for encoding plans. We would like to retain this simplicity in any planning system implementation. Analysis is easy since possible behaviours correspond to *paths* in the state-space structure. Such paths are trivial to recognise, and easy to generate. Of course, generating the *right* path is anything but easy.

In what follows, we refer to the reasoning about what does and does not hold at various points within a plan as *question-answering*, following Tate (1976). A procedure which implements this question-answering reasoning will be called a question-answering procedure (reasonably enough). The term question-answering seems appropriate, since much of the work involved with linking an operator into a plan amounts to asking questions of the form: "is p the case?" with respect to a given location in the plan. If we assume for now that the procedure cannot suggest structural plan modifications, then if p does not contain variables, the answer can be "yes" or "no". If p does contain variables, the answer can be "yes, with the following variable substitutions", or "no, under any variable substitutions". If the the question-answering procedure is allowed to suggest plan modifications, then the answer might be "yes, if you make the following changes to the basic plan structure". Such an option is followed by the Nonlin (Tate, 1977) question-answering system, when it suggests a new link to guarantee the truth of an assertion at a given point in the plan. Whether or not it is permitted to suggest structural plan modifications, such a question-answering procedure forms the basis for what we shall refer to as a *plan decision procedure*. This term is used to indicate a procedure which accepts a plan and a goal state specification, and can decide whether the plan solves the problem, or whether the plan is deadlocked, and should be abandoned. Thus a plan decision procedure is the means by which a plan is signalled as being worth further consideration, or is determined to be uninteresting. In this latter case, the plan will often be scrapped in favour of some other alternative.

3.2.2. Action-ordering plan representations.

It is possible to represent a plan as something other than a state-space structure. Plans can come in various forms. For instance, plans can be represented as Lisp programs, as done by Sussman (1973) in his Hacker system, or plans can be represented in an interval logic, as done by Allen and Koomen (1983), Cheeseman (1983), and Kowalski and Sergot (1984).

A common form of plan representation is a set of actions with constraints on the order in which the actions can be executed. For example, it is often convenient to represent a plan as a simple list of actions. The actions are to be executed in the order in which they occur in the list, beginning with the first action executed in the initial state. This representation is not state-space. It consists of actions and ordering relations on those actions. We refer to the general class of representations for behaviour which use actions and ordering relations on actions as *action-ordering* representations. Lansky (1985) calls these representations "behavioral", in the sense that they focus more on the actions that are to be performed, and less on the assertions that each action affects. We avoid the word "behavioral" (or even "behavioural") in this context, and use the less emotive term "action-ordering" instead. Notice that a plan as a list of actions is an action-ordering structure in which the order on actions is total. Ordering information is expressed by the relative position of any two actions in the list.

We turn now to an examination of one of the more successful action-ordering representations: Sacerdoti's (1975) procedural net.

Sacerdoti's planning system, NOAH, used the procedural net as a representation for plans. NOAH and its descendents, such as Nonlin (Tate, 1977), Deviser I (Vere, 1981), SIPE (Wilkins, 1983), O-Plan (Currie and Tate, 1985), Tweak (Chapman, 1985), Forbin (Miller, Dean, & Firby, 1985), and Deviser III (Vere, 1985), all use plan representations based on Sacerdoti's original idea. The following comments are expressed principally in terms of the NOAH system, but apply equally to these newer planners.

A *procedural net* has been defined as "a network of actions at varying levels of detail, structured into a hierarchy of partially ordered time sequences." (Sacerdoti, 1975, page 10). For

current purposes, the main interesting feature of the procedural net is the fact that its basic objects are actions, and ordering relationships on those actions. A net can be drawn as an action-on-node graph, with directed arcs between nodes. An arc running from one node α to another node β means that the action denoted by α must occur *before* the action denoted by β.

An action-ordering representation such as the procedural net contains no explicit notion of *state*. Any given action-ordering plan can be considered to be in any one of some number of possible states, but the state which it is in is not *explicitly represented* in a single location in the net, as in a state-space representation. For example, it is impossible to point to a particular location in a procedural net and say, "ah yes, there we have a possible state of the world"; in a state-space representation, this *is* possible. This lack-of-state in a procedural net can cause problems in net construction. It is possible however, to derive a state-space account of a procedural net, and answer questions about what holds or does not hold with respect to this account. We discuss how this can be done in section 3.5, below.

An action-ordering representation is often convenient, since it does not force us, when constructing a plan (either manually, or automatically), to give the assertions that each action affects. We can simply order two actions, and need not say what "state" will obtain when the first finishes, and the second begins. As noted by Lansky (1985):

> ... strictly state-based approaches to domain description can be awkward in describing behavioral properties; i.e., those that entail complicated causal and temporal relationships *between* actions. Priority requirements, for example, fall into this category; they restrict future relationships between actions (for example, the order in which a service is performed) based on past relationships (the order in which requests for service were registered).
>
> (Lansky, 1985, p.6).

Many people now feel that the main strength of the procedural net is its ability to describe parallel activity. However, the procedural net was originally introduced by Sacerdoti "with the intention of executing the actions serially" (Sacerdoti, 1980, p.4). The procedural net was actually introduced to allow certain *least commitment* (with respect to action ordering) plan construction techniques to be employed. It turns out that if by "least commitment" we mean the policy of leaving causally independent actions un-ordered, then the procedural net actually offers nothing more than certain sorts of state-space structures.

We can construct a state-space structure able to describe parallel actions as follows. As before, we use a graph to model the state-space, but we now allow sets of actions to label the arcs. The understanding is that in order to traverse an arc, we must select and execute each of the actions in its labelling set. For example, say that some particular set contains actions *1*, *2*, and *3*, and that an arc labelled by the set joins two states, *A* and *B*. Then the understanding will be that if in state *A*, we can arrive at state *B* by executing *1*, then *2*, then *3*, or by first executing *2*, then *1*, then *3*, and so on. The order doesn't matter: as long as all the actions are executed, the resulting state will be *B*.

So we can see that the ability to leave causally independent actions un-ordered is not an ability exclusive to the procedural net. If properly defined, state-space representations can do this as well. For an example, see Reisig (1985, p.28), where state-space structures are defined which allow sets of actions to label arcs joining states. We use such structures in chapter 4, when defining the projection (forward simulation) of a plan.

We have said that the procedural net is an action-ordering representation. Until something is said about the nature of the ordering relation in an action-ordering plan, the ability of the plan to describe parallel activity is undetermined. If the order on actions in the plan is total, then we have only serial action execution, but still operate within an action-ordering representation. A total order is essentially what obtains when a plan is encoded as a list of actions.

Above, we claimed that the main feature of the procedural net was the fact that it is an action-ordering plan representation. Sacerdoti's dissertation (1975), also stresses that the procedural net contains a notion of *hierarchy of action*. This idea is quite central to the way that procedural net derivatives have controlled the planning world for the last ten years. We do not address the issue of action hierarchies here. This is fair, since the basic nature of a plan representation (state-space or action-ordering) and the ability of the representation to describe hierarchical actions are independent. In chapter 7, we return to the problem of integrating Sacerdoti's insight on representing actions in abstraction hierarchies into the representation for plans developed in this dissertation.

It was suggested earlier that state-space representations are desirable because of the way they simplify plan construction reasoning. This reasoning must be embodied in the question-answering procedure, which can answer questions about what assertions hold in specified parts of a plan. It seems relevant to ask how the question-answering procedure is affected if an action-ordering plan representation is used.

A question-answering procedure can operate over an action-ordering representation in one of two distinct ways. The procedure can be constructed so as to derive a state-space structure from the action-ordering representation, and answer all questions with respect to this state-space account. Answering a question then becomes as simple as in the case of a state-space plan. However, it is often better to perform question-answering functions on demand, without ever generating a state-space structure. This latter combination of action-ordering plan representation plus on-demand question-answering can provide the simplicity of analysis afforded by a state-space representation, without the (often considerable) space requirements brought on with the maintenance of an explicit state-space structure.

If desired, it is very simple to create a state-space structure for question-answering purposes. For example, suppose that we have a plan represented as a simple list of actions; that is, the order on actions in the plan is total. To create a state-space account of this plan, one need only start with the initial state (as given in the problem), and create successor states by applying the actions in their list-order, according to the actions' add-list and delete-list specifications. Question-answering on demand is equally easy: given an assertion p, and asked whether p holds (say) after the execution of the last action in the plan, one need only look back along the list of actions, attempting to find an action whose add-list or delete-list contains p. If an add-list is found, the answer is "yes"; if a delete-list is found, the answer is "no". If no action in the list has p on either list, the answer is determined by looking up p in the initial state. Of course, it may be necessary to unify a variable in p with a constant, generating a variable substitution.

From this example, it seems rather easy to perform question-answering functions on an action-ordering plan. But in this example, the order on actions is total; that is, only serial

EX BIBL. UNIV. EDINBURGH.

execution is possible. When the order is partial, i.e., when the plans may be parallel, this question-answering becomes slightly harder. We return to this issue in section 3.5 below, since it holds a key position in our critique of the procedural net.

3.2.3. Summary.

We now briefly summarise the comments of this section, and use this summary to justify definitions of the terms *linear plan*, and *Nonlinear plan*.

There are two main "types" of plan representation that have seen frequent service in the planning community. First, we have the state-space representations, as borrowed from the classical representations of physics and other domains in applied mathematics. A state-space *plan* contains descriptions of states of the world the plan is to be executed in, and move operators which correspond to the actions to be carried out to execute the plan. State-space plans are mechanically easy to construct, due to the way that state descriptions "package up" related information. A state-space solution is trivial to recognise: it is quite simply a *path* from some initial state to a state satisfying the goal specification. It is never hard to answer questions about what is or is not the case with respect to a given state, and this makes plan construction *technically* simple. Of course, creating a correct plan quickly from various alternatives proves rather combinatorially ugly.

There are also action-ordering representations, of which the "list of actions" plan is a simple case, and the procedural net the most oft cited example. Action-ordering plans allow us to describe the relationships among actions directly, rather than through states and predicates contained within states. Action-ordering plans suffer since it is not always obvious what "states" of the world are possible, given an initial state and an arbitrary plan. For this reason, plan construction algorithms can build "unexecutable" plans, unless they are extremely careful. (See Charniak and McDermott, 1985, pp. 499-512, to gain an appreciation of the difficulty of automatically assembling an action-ordering plan.) Unlike a state-space representation, a possible behaviour in an action-ordering plan does not correspond to a simple path in a graph. This can make plan creation and solution recognition quite difficult for the unwary. It can often take

extensive analysis to realise that a given plan is a solution to some specified problem.

We have tried to show that the ability of a representation to describe parallel activity is independent from whether it is state-space or not. That is, some state-space representations can describe parallel activity, and some cannot. Some action-ordering representations can describe parallel activity, and (once again), some cannot. If we define our state-space representation carefully, we can allow sets of actions to label the arcs joining states, and execute state-transitions by selecting and executing all the actions in the relevant set. Action-ordering plans, of which the procedural net is a prime example, may or may not be able to describe concurrent activity, depending on the type of order imposed on the actions in a plan. In the case of the procedural net, the order is partial, and thus actions can be executed concurrently. With a plan as a list of actions, the order is total: only serial action execution is possible.

This section has also discussed the issue of question-answering in terms of plan representation. In the case of state-space plans, the procedure is simple: look up the required assertion in the specified state. Things are slightly harder with arbitrary action-ordering representations. A simple example was given for a totally ordered plan: look backward from the specified point along the list for an action which references the assertion. If an add-list is found containing the assertion, the answer is "yes", and if a delete-list is found, the answer is "no". If no action can be found which references the assertion, the answer is determined by looking for the assertion in the initial state (as given in the problem).

Henceforth, we reserve the term *parallel* to refer to those plans that describe potentially concurrent activities. This means that we include state-space representations in which arcs can be labelled with sets of actions, and action-ordering representations in which the "before" order on actions is partial. Using either type of representation, actions can only legitimately be executed in parallel when their pre-conditions and post-conditions are disjoint. (This is formalised in chapter 4, definition 13.) If a set of actions share no pre- and post-conditions, then they are causally independent, and can be executed in any order: we need not reason about the way in which the actions may be interleaved. This sort of parallelism is concerned with the potential parallelism of

action execution; actions can be executed in any order, or even concurrently.

If a plan does not describe parallel activities, we say it is *serial*. It is not uncommon in the planning literature to use the term *Nonlinear* as we have used *parallel*, and *linear* as we have used *serial*. We avoid these terms since they often impose confusion through etymological similarity to the word *linear*, as it occurs in the phrase *linear assumption*.

In this section, we have addressed issues of plan representation. In the next, we turn to plan construction, and discuss the nature of the planner's search space. This is done so that we may come to grips with the somewhat inappropriately named "linear" assumption.

3.3. Searching for plans.

In this section we examine two related issues: the nature of the search space, and the so-called "linear assumption". This is done to resolve some confusion in the literature on what parallel plans have to do with the linear assumption. We begin by examining the nature of the planner's search space. Two alternatives are distinguished: a space of world states, and a space of partial plans. It is shown that if a planner searches a space of world states, then it has conflated its plan construction reasoning and its search-space control. This results in plan modification being possible only at the tail-end of the plan, using chronological backtracking. A better option is to search through partial plans. This allows reasoning to operate over the entire plan, and allows the planner to perform plan modification at any point in the plan. We use these comments to argue that parallel plans do not solve or even address the "linear assumption".

3.3.1. The search space.

First, we examine the nature of the planner's search space. Problem solving is often viewed as search. Solving a problem using the search metaphor can be understood as follows.

> We postulate some kind of space in which treasures are hidden. We build symbol structures (nodes) that model this space, and 'move' operators that alter these symbol structures, taking us from one node to another. In this metaphor, solving a problem consists in searching the model of the space (selectively), moving from one node to another along links that connect them until a treasure is encountered.
>
> (Simon, 1983, p.7)

It is interesting to consider the nature of the nodes within the search space. Exactly what do they describe? As noted by Charniak and McDermott (1985), and Tate (1985), the nodes are often interpreted either as describing *states of the world* (task domain), or as *partially completed plans*. When describing a Nonlin-like (see Tate, 1977) planner (derived from NOAH; see Sacerdoti, 1975) which produces task networks (a form of action-ordering representation, derived from the procedural net), Charniak and McDermott say that:

> The space searched by our planner is the space of 'partial plans,' or task networks. Operators reduce tasks to subtasks, and impose orderings on hitherto unordered tasks. A goal state is a plan that is guaranteed to work without any protection violations.
>
> A rather different use of search, and historically prior to Nonlin, is to search through world situations rather than through the space of alternative task networks. Under our stringent assumptions about time, we can think of planning as a search through possible states of the world. We can take the initial world model as the initial state; we can take primitive acts (described by addlists and deletelists) as operators; and the state of affairs to be made true (achieved) in the final world model as the goal description.
>
> (Charniak and McDermott, 1985, p.514).

There are thus two sorts of search space that a planning system can explore. In one, the nodes examined contain descriptions of possible states of the world, and in the other, the nodes contain possible partial plans. We will refer to the former as search through world states, or simply state-space search, and to the latter as search through partial plans, or partial-plan search.

How does this distinction relate to the comments of the last section? The issue is really one of *reasoning*. A planner which searches through world states can also have plan structures in each node of its search space. However, such a planner can only use each node's world-state information to guide its next move in the search space. If the planner does carry along a plan in each search space node, then it is natural to think of the plan as a path describing how the planner arrived at the node containing the plan. Thus the state information in the node describes the world state that will exist at the termination of the plan. Retaining this path information might be a good idea, since if the plan is left implicit in the planner's state-space search graph, it may be hard to extract on termination. Of course, it may not be impossible, just very hard. If each search-space node contains not only world-state information, but as well, plan (or path) information, then when a node satisfying the goal state specification is found, the path contained in the node describes the plan to achieve the state that has been found. The plan describing the path

may be in any form, and be arbitrarily complex.[1]

Recall that a path in a state-space *plan* is basically a plan simulation, or projection of a domain behaviour. When a planner searches through possible world-states, it is using its state-space navigation procedure to do its plan construction reasoning. This allows the planner to essentially avoid any problems with question-answering procedures, since all questions are asked with respect to the "current" search-space node. The answer is determined by a simple set membership check: if the assertion is in the node's database, then "yes", otherwise "no". Basically what happens when a planner searches through world states is a conflation, or merging of two distinct things: plan construction reasoning, and search space navigation. The construction of a plan and the search for it become identical.

The discussion in section 3.2 suggested that the reasoning required to build state-space plans is simple, due to the way that states package-up related information. In practise, such plans rarely exist explicitly in a planner. Instead, the planner will search through a space of world states, and use the state-space structure of its search-space to perform this packaging function for it. There are however, some explicit state-space plan representations in use; most notable are Georgeff (1984), and Georgeff, Lansky, and Bessiere (1985), and Georgeff (1986). We discuss Georgeff's work more fully in section 3.4.13, below.

Why would a planner perform search through a space of world states? It's because the search procedure is all the *reasoning* that's required of the planning system. The reasoning required to build a plan is packaged into the planner's next-state generator. This means that the planner will only generate *possible* plans, and need perform no checking after extending a plan. Implementing such a plan construction strategy is also often extremely easy, given the appropriate language and programming environment. If the language and environment support some sort of *context* mechanism together with a facility for chronological backtracking, then a solution which calls for searching through world states is easy to implement. Each context can be made to

[1] STRIPS can be thought of as operating in this way. It searches through a space of world states to find a plan, but encodes the plan it finds as a "triangle table". The triangle table format provides the plan execution system (Planex) with all the information it requires in compact form.

contain a number of assertions about the world, and new contexts created by applying domain operators. When a context is reached in which no further operators are applicable (corresponding to a world state from which no progress is possible), the language's backtracking mechanism can be invoked to restore a context which contains a previously ignored option. When a context is derived which contains the required goals, the search can stop. Often, this is what happens if languages such as Conniver (McDermott, 1974), and Planner (Hewitt, 1972; Sussman, 1971), or Prolog are used to build problem solving systems. Of course, the languages do not *force* this on a programmer. However, given the available tools, such an implementation often appears the obvious choice, and certainly involves the least programming effort.

What of the other sort of search space, that of partial plans? If a planner does this, we can consider it to be *separating* its search-space exploration procedure from its plan construction reasoning: the search for a plan and the plan construction process become distinct. Instead of having the plan construction reasoning encoded in the planner's search-space navigation procedure, we will have plan construction reasoning operate over a plan contained in each search-space node. The results of the reasoning will guide the search through the space of possible plans. In terms of the discussion in section 3.2, above, the planner must apply its plan decision procedure to any given plan under consideration, in order to decide what to do next. This procedure must announce when plans are un-executable, and when successful plans have been found.

If a planner searches through partial plans, it can add and delete actions at various points in the plan. When the planner searches through world states, plan modification is possible only at the "tail end" of the plan. Actions are added to the plan by trying out another operator application, and operators are deleted from the plan when backtracking occurs. This is a result of conflating the plan construction reasoning with the search-space navigation control. However, if the planner searches through a space of partial plans, it can add and delete actions wherever in the plan it sees fit.

If a planner searches through partial plans, then the move operators in its search space correspond to planning operators. The operators available will depend on how the planner

proceeds. For instance, there may be an operator for adding an action to a plan, and another operator for ordering two previously unordered actions. If we were interested in meta-planning, we would need operator schemas which described these possible actions in the search space. However, here we do not consider the possibility of meta-planning. We agree with Wilkins (1983), when he says that the entire issue of meta-planning is ill-understood. Object-level planning must be understood before attempting to apply planning techniques to more difficult meta-level problems. If a planner searches a space of partial plans, we should not yet demand that it have a set of operators which describe movement possibilities in its search space. For now, it will suffice to encode the possibilities for search space movement in the program which implements the plan construction algorithm.

If a planner searches through a space of world states, then it will have no problem deciding when it has found a plan. Recall the discussion of the last section. State-space structures are easily managed, since solutions are simply paths through the space. If a planner is using its state-space search structure to encode its plan decision procedure, it wins due to the simplicity of state-space analysis. However if the planner searches through partial plans, then it must have a decision procedure for deciding when it has found a successful plan. The reasoning will likely be different for each sort of representation. Various representations for plans are possible, and it seems reasonable to suggest that the plan reasoning required will vary as a function of the representation used.

If the plan representation is state-space, then the reasoning is (once again) particularly straight forward. Plans must be thrown out, and backtracking in the search-space of plans initiated, when paths *in a plan* expire unsuccessfully. Backtracking means restoring the plan under consideration to an earlier state; in particular, to an earlier plan state from which there exists a previously unconsidered plan modification option. The search can terminate when the plan decision procedure announces that a successful plan has been found.

The same thing must happen if the planner is searching through partial plans, using an action-ordering representation. The only difference will be in the planner's reasoning which

detects dead and successful plans. In section 5.5, we suggest that an easy way to reason about an action-ordering plan is to create a state-space account of it, and answer all questions with respect to this state-space account. This technique isn't the only one, just easy and obvious. Chapman (1985) has formalised the conditions under which an arbitrary action-ordering plan is acceptable, and has shown how one can build a non-deterministic plan construction procedure from it.

3.3.2. The linearity assumption.

We now briefly address the "linear assumption" in order to discuss the relationship between this assumption and parallel plans.

While a planner moves through its search space, it is attempting to solve a problem: if there is no problem, then there is no reason to search for a solution. A prerequisite for solving a given problem is having a plan which, under some sort of simulation, creates a situation satisfying the goal specification. If the planner is clever, it will make reference to its goals when creating the plan, since un-constrained forward search is explosive in any realistic domain. All searching should thus be done in a goal-directed way, by finding out what is required by the problem, and by then attempting to construct a plan which can produce a situation which satisfies the goal specification. It is this issue of goal-directedness that has demanded so much attention in planning research over the last twenty years.

One of the major contributions to problem solving by the GPS[2] project (Ernst and Newell, 1969) was the technique of means-ends analysis, abbreviated here as *MEA*. MEA is a search heuristic which calls for applying operators that can be shown to be relevant to reducing the differences between the initial and goal states, as specified in the problem description. MEA reduces differences, one-by-one, until no differences remain. Unfortunately this technique will not always arrive at a solution. For instance without some sort of help, MEA cannot find a solution to the problem of swapping the contents of two registers. The issue is well described by Simon (1983):

---

[2] GPS stands for "General Problem Solver", a title that is now understood to be rather over-ambitious.

> The search procedure of GPS is built on the implicit premise that if the present situation differs from the goal situation by features *A,B,C,...*, then the goal situation can be attained by removing the differences *A,B,C,...*, in some order. Of course this premise is false unless the matrix of connections between differences and operators can be triangularized. This matrix can be triangularized just under those conditions when an appropriate composition axiom would be valid in the modal logic; that is, just when there is independence among the actions.
>
> (Simon, 1983, p.15).

Earlier, Sussman (1973, p.58) had expressed the same idea, and had called it the *linear assumption*: "Subgoals are independent and thus can be sequentially achieved in an arbitrary order.". Here, we refer to this as the *linearity assumption.*

It is often thought that parallel plans "solve" the linearity assumption.[3] This is entirely wrong. The linearity assumption has to do with the way that a planner constructs its plans, and not with the plan representation itself. The word "linear", as used in the phrase "linear assumption", refers to the way that a planner approaches plan construction: a "linear" planner will select a goal from the top-level goal set, and attempt to achieve that goal from the initial state. If the goal can be achieved, then some next goal is selected, and an attempt is made to satisfy it, proceeding from the state created to satisfy the first goal. And so on. Such a planner proceeds *linearly* in the sense that it only considers different linear orders of *goals* from the top-level goal specification.

As some researchers have pointed out, GPS-style MEA works best when operators do not re-introduce large differences already reduced. (See for instance Charniak and McDermott, 1985, p.305.) If a set of operators have this "non-interference" property, then MEA will generally work well on any problem in a domain to which the operators are applicable. It is in this sense that Simon (quoted above) uses the word "independence", and Sussman uses the word "independent". Operators are independent if in general, they allow a solution to monotonically approach the goal state, reducing differences one-by-one without re-introducing already achieved goals. Operators for many sorts of problems have this property, if written correctly. For instance the problem of solving systems of algebraic inequalities seems well suited to the application of an MEA-type heuristic. However it appears impossible to write non-interfering operators for even simple

[3] The literature is full of statements which perpetuate this confusion. For two representative views, see Barr and Feigenbaum (1982, p.520), and Stefik (1981, p.134).

planning domains. The blocks-world is a good example: each operator *must* re-introduce differences which may have been previously reduced. Operators simply move blocks about, and *introduce* just as many differences as they *reduce*.

We have discussed the linearity assumption only to make it clear that parallel plans do not allow a planning system to do any better. The linearity assumption has to do with plan *construction*, and is embodied in MEA-like heuristics which attempt to achieve goals in a direct fashion. Parallel *plans* are those which contain causally independent actions. Such actions can execute concurrently. Planning systems which employ parallel plan representations must still address the issue of the order in which they will pursue their goals. Recent work by Vere (1985) has begun to make this clear. We have stressed the issue here to ensure understanding of how the parallel (some might say "Nonlinear") plan representation developed in chapter 4 relates to the linearity assumption.

The major contribution of this work is in the area of plan representation. The representation for plans we present is an action-ordering parallel one. We do not address the linearity assumption.

3.3.3. Summary.

We now briefly summarise the comments in this section.

Planners must search to find correct plans. Two sorts of search space can be distinguished: a space of world states, and a space of partial plans. If a planner searches through world states, then it has conflated its plan construction reasoning with its search space navigation. A better idea is to search through partial plans. This allows a planner to add and delete actions at various points in a plan.

Since planners search to find plans, it seems a good idea to make the direction of search depend on the goals to be achieved. Means-ends analysis is one heuristic which does this. MEA calls for picking out goals one-by-one from the goal specification, and solving each one "directly". The first goal attempted is solved from the initial state. The second is solved from

the state which results from the solution of the first goal. And so on. Sussman has said that a planner which does this is making the "linear assumption". We renamed this the *linearity assumption*, and argued that it has absolutely nothing to do with whether a plan is serial or parallel. The linearity assumption is about plan construction. The terms *serial plan* and *parallel plan* describe certain types of plan representations: serial plans are those in which no parallel execution is possible. Parallel plans contain actions which can be executed concurrently. *Parallel plans do not help a planner which makes the linearity assumption.*

3.4. What have planners done in the past?

We now use the comments of the last two sections to analyse some of the previous major planning systems. Recall the features that we have itemised. Plan structures are often either state-space or action-ordering. Either sort of representation may or may not be able to represent parallel activity. When constructing a plan, a system must search through some sort of space. The nodes in the space searched may correspond to possible world states, or to partially completed plans. To make the search more efficient, a planner may employ a search heuristic, of which various are available, MEA being a common choice.

In the following sub-sections, we use these parameters to classify previous major planning systems. We first give a very brief description of the relevant features of a system, and then give a point-form account of the system in terms of the parameter set distinguished above. Some amount of mis-representation is perhaps inevitable with any "rational reconstruction" such as this, but the task still seems worthy of being undertaken.

3.4.1. Green's system.

Green (1969) formulated planning problems in the predicate calculus, and used a resolution theorem prover to produce plans. In Green's system, each assertion included a state argument, which described the state in which the assertion held. (Perhaps some still hold to this day, who can say?) The idea of including such a state argument in assertions comes directly from McCarthy (1958, 1969). This technique is often referred to as the *situation calculus*. In Green's

implementation of this idea, a goal is a formula with an existentially quantified state variable. The system attempted to prove that there exists a state in which the required assertions are true (a state in which the specified assertions hold).

The performance of actions and the states thus produced were considered equivalent. In Green's system, the function *do* mapped from states to states. Action performance and states were represented by structures of the form:

do(action(...), S)

where the ellipsis denotes the arguments of the action, and S is another such structure, or the primitive (initial) state. For instance, *on(a,c,do(trans(a,b,c),s0))* says that block *a* will be on block *c* in the state which results from executing the action denoted by *trans* in the state denoted by *s0*. A set of assertions were true in the same state by virtue of their common state-ancestry; that is, all assertions of the form *pred(args,S)*, where *pred* is some predicate, *args* are the predicate's arguments, and S is common to all assertions, were all considered to hold in the same state.

Movement operators were given as implications. A frame axiom was required for each predicate that each movement operator did not affect. Thus, the number of frame axioms required was the product of the number of predicates and movement operators (i.e., large).

System: QA3 theorem prover.
Plan structure: action-ordering.
Parallel/Serial: serial.
Search space: world states.
Search control technique: as used in the QA3 system for selecting resolvents.

3.4.2. Kowalski's system.

Kowalski (1979) offered a different formulation for planning using the predicate calculus. Kowalski used a predicate *holds* to indicate that a given assertion holds in a given state. For instance, *holds(on(a,c),s0)* is used for Green's expression *on(a,c,s0)*. This means that the number of frame axioms required is equal only to the number of action descriptions, since it is possible to quantify over all other assertions not affected by an action, using the *holds* predicate. For discussion purposes, we present a tiny Prolog implementation of Kowalski's system in figure 3.1.

In the example given, there is only one movement operator and frame axiom.

Once again, a goal state was specified as a conjunction of assertions containing an existentially quantified state variable. Actions and states were of the same form as in Green's system, but instead of maintaining the explicit state-space information regarding the holding of each predicate, Kowalski's system only looked up the information when required. In the example, this must be done since the predicate *poss* returns a possible *plan* each time it is backtracked over, and this plan must be examined to determine what does and does not hold at its end. Instead of having a collection of assertions of the form *pred(args,state_spec)* for each *pred* true of *state_spec*,

```
poss(s0).          /* Initial State is the first possible one */

holds(on(c,a), s0).        /* Initial State specification */
holds(on(a,t1),s0).
holds(on(b,t2),s0).        /*      c                      */
holds(clr(c), s0).         /*      a       b              */
holds(clr(b), s0).         /*      t1      t2      t3     */
holds(clr(t3), s0).

holds(clr(Y), do(trans(X,Y,Z), S)). /* Action postconditions */
holds(on(X,Z), do(trans(X,Y,Z), S)).

pact(trans(X,Y,Z), S) :-           /* Action preconditions */
        holds(clr(X), S),
        holds(clr(Z), S),
        holds(on(X,Y), S),
        X\= Z, Z\= Y, X\= Y.

poss(do(U,S)) :- /* Defines possible action sequences */
        poss(S),
        pact(U,S).

holds(V, do(trans(X,Y,Z), S)) :-  /* Single action frame axiom */
        holds(V,S),
        V\= clr(Z),
        V\= on(X,Y).
```

Figure 3.1: A simple Prolog implementation of Kowalski's formalism.

we have *state_spec* as a structure we can examine to find whether or not required assertions hold. In figure 3.1, this lookup function is performed by the predicate *holds*. Essentially, this predicate performs what we have been calling question-answering.

System: untitled.
Plan structure: action-ordering.
Parallel/Serial: serial.
Search space: partial plans.
Search control technique: breadth first on the length of the plan.

3.4.3. STRIPS.

STRIPS is probably the single most significant planning project to date. No short description can ever adequately describe the contributions and impact of the STRIPS system on the planning world. However, we shall feel free to try.

STRIPS was an elegant adaptation of the General Problem Solver (Ernst and Newell, 1969) to planning problems. It included an implementation of means-ends analysis based on the QA3 theorem prover. This use of a theorem prover is quite surprising, since STRIPS also introduced the idea of using non-logical state-transforming operators. The nature of these operators, including the idea of using precondition-lists, add-lists, and delete-lists, is discussed in chapter 2.

As far as structure goes, STRIPS was innovative in its basic plan representation. It employed Triangle Tables to encode all the essential features of a plan's teleological structure. This information was supplied to the the Planex system, to enable accurate runtime plan execution monitoring. Triangle tables are currently experiencing a minor resurgence of interest as a programming language for robot actions (Nilsson, 1985). However, the representation for plans developed in chapter 4 goes beyond what triangle tables have to offer.

STRIPS is one of the classic systems to apply the linearity assumption, as embodied in means-ends analysis. STRIPS searched through a space of world states to find a solution, and employed MEA to guide its way. As a direct result, there are many problems for which STRIPS can provide no solution. The register exchange problem is typical of the sort of problem STRIPS was unable to solve.

System: STRIPS.
Plan structure: action-ordering (teleology used only for execution monitoring).
Parallel/Serial: serial.
Search space: world states.
Search control technique: means-ends analysis.

3.4.4. Hacker.

Hacker (Sussman, 1973) was designed primarily as a learning program, but also contained a planning element. The learning was of a "procedural" sort; that is, the program learned new "skills". Hacker proceeded by creating a plan from its plan library, and submitting this plan to its "gallery of critics". After criticism, Hacker assumed that the plan would work. The plan was then "simulated", and bugs detected. The bugs were analysed, and the result of this analysis passed on to the learning element to enable Hacker to do better on its next attempt.

In order to help the bug detection phase operate, each plan had attached to it an account of its teleological structure. This account indicated where in the plan goals were achieved, and where they were required. The bug detector used this information to index into a library of bug types. Each bug type was in essence a "teleological template", and could be matched to any given plan to diagnose that plan's bug. As will become evident in our discussion of various planning systems, teleological information is of vital importance in constructing plans. Hacker was the first program to make use of such information during plan construction.

Hacker searched through a space of partial plans, and used its critics to suggest structural plan modifications. The critics were un-constructive, in the sense that they found reason for existing faults, and were not able to suggest fixes in advance of fault appearance. The interesting feature of Hacker as a *planner* is the way it turned attention to the issues of having a plan as a structure, and using procedures to reason over the structure, modifying it as necessary. Thus, it was one of the first planners to actually search through a space of partial plans. It was also in this project that the "linear assumption" was first acknowledged. In making this assumption, Hacker greatly reduced the size of its plan search-space. However in making this reduction, it lost the ability to solve many interesting problems.

The plan representation used by Hacker was essentially action-ordering. The plans were encoded as Lisp programs, but no interesting programming constructs could be handled. Only serial program execution was ever considered. For this reason, it seems reasonable to say that Hacker did actually employ an action-ordering plan representation.

System: Hacker.
Plan structure: action-ordering with teleology add-on.
Parallel/Serial: serial.
Search space: partial plans.
Search control technique: try then debug; made linearity assumption.

3.4.5. Interplan.

Interplan continued where Hacker left off. Tate (1975) used a perspicuous representation for the teleological structure of each plan, called a *tick-list*. The tick-list allowed Interplan to pursue goal reorderings not considered by Hacker. An example of a problem which Interplan could solve, and Hacker could not, is shown in figure 3.2. This problem is one that cannot be solved by a planner that makes the linearity assumption.

Once again, Interplan searched through a space of partial plans; however it used a procedure to guide its search which opened the space to include a solution to the problem of figure 3.2. This procedure performed *subgoal promotion*: this can be viewed as a means of relaxing the linearity

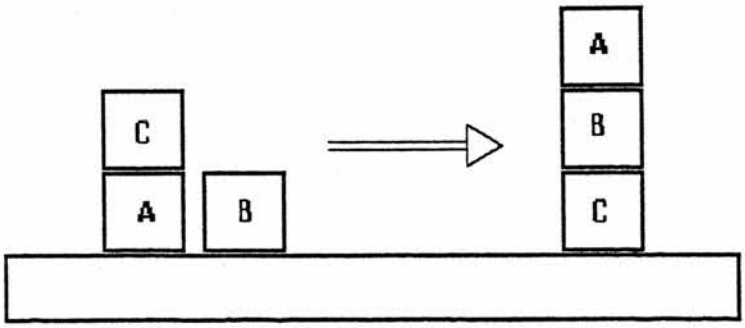


Figure 3.2.

assumption, by attempting permutations of not only the highest level goals of the problem, but as well, considering *relevant* goal interleavings which include interacting sub-goals introduced during plan construction.

Interplan was interesting because of the way it highlighted the fact that it is possible to reason about a very simple teleological account of a plan, with significantly improved problem solving ability. It was the first planning system to use teleology to actually correct for interactions between parts of a plan initially assumed to be independent.

System: Interplan.
Plan structure: action-ordering, with teleology add-on.
Parallel/Serial: serial.
Search space: partial plans.
Search control technique: teleologically relevant subgoal promotion.

3.4.6. Waldinger's system.

One of the problems of the Interplan system was the way it threw out much of the work done in pursuing any unsuccessful plan. Once a given goal-order was found not to work, the plan was thrown away, and work began anew on a plan to achieve goals in the new order (in Interplan, this order was called an "approach"). Waldinger (1977) suggested a technique called *goal regression* which overcame this problem. Goal regression calls for building a plan to achieve one of the top-level goals, and then modifying that plan to achieve the other top-level goals as well. A plan is modified with respect to a goal by finding a place in the plan where the goal can be achieved or is found to be compatible.

The basic idea amounts to "dragging a goal back" along the list of actions that is the plan, until some inter-action gap is found in which the goal can be inserted. Goal regression modifies plans more constructively, and less wastefully than Interplan, since existing plans to achieve goals are not thrown away when a goal achievement attempt fails.

Once again, this technique can only be applied when a planner searches through a space of partial plans. Goal regression will call for trying a goal at various points along the plan, until a location is found which works. The search space considered by Waldinger's system excluded

movements in which an action was deleted from a plan. Once an action was added it was in for good or bad and would never be removed. Given a plan of length *n*, the technique would only call for immediate consideration of plans of length *n+1*. Although plans of length *n-1* should have been candidates for consideration, they weren't.

A system employing this technique must maintain each plan as a list of actions, since the state which will obtain following the execution of any given action will be a function of not only the action, but as well, a function of those actions that precede it. Since this list is liable to change, state information can't be static. This means that question-answering must be performed on demand, as required.

System: untitled.
Plan structure: action-ordering.
Parallel/Serial: serial.
Search space: partial plans.
Search control technique: goal regression.

3.4.7. Warplan.

Warplan (Warren, 1974) is based on a technique much like Waldinger's goal regression. Warplan used *action regression* to construct plans. This technique focuses on the actions required to achieve goals, rather than the goals themselves.

Given a partial plan and a goal, Warplan finds an action to achieve the goal, and then regresses the action along the plan to a point at which the action can be peacefully inserted. Peaceful insertion means that the action's preconditions can be simply achieved at the selected spot. Warplan suffered from the problems itemised above for Waldinger's goal regression, and in addition, would often fail to recognise when an action being regressed ceased to be necessary.

Warplan seems most noted for its size and efficiency. It was one of the first planners written in Prolog. Since Warren included the Prolog source-code in the research report describing the system, the planner has been received quite well, and re-implemented many times over.

System: Warplan
Plan structure: action-ordering.
Parallel/Serial: serial.
Search space: partial plans.
Search control technique: action regression.

3.4.8. NOAH.

NOAH (Sacerdoti, 1975b) is possibly one of the most misunderstood (and thus misrepresented) planning systems ever. It is often claimed that the procedural nets used by NOAH "solve" the linearity assumption. We argued above that this is not so. The careful reader of Sacerdoti's original dissertation (1975a) will notice that no such claims are made by Sacerdoti himself.

NOAH as a planner did not in fact make the linearity assumption. But as discussed above, this had nothing to do with its plan representation; rather, not making the linearity assumption was a direct result of NOAH's plan construction algorithm. NOAH reasoned "backwards", starting from its goals, and extending its plan towards the initial world state. For this reason, all goals attempted by NOAH were relevant in any eventual solution it derived. The linearity assumption says that there exists an order of goal pursuit, beginning with some goal being achieved from the initial state. NOAH did not construct its plans in this manner, and thus did not need to make the linearity assumption. This kind of plan construction is possible with almost any sort of plan representation. The issue of goal ordering is not addressed in NOAH, since it has no mechanism for backtracking out of a bad choice.

The bulk of NOAH's importance stems from its plan representation: The procedural net. These nets were important at the time for two reasons. First, they contained an explicit notion of action hierarchies; and second, they were the first plan structures which allowed "least-commitment" plan construction polices to be adopted.

An action hierarchy allows an action to be represented by a single node at one level of a net, and expanded into more nodes at a lower level. This expansion corresponds to analysing the action in greater detail. Such a technique permits the construction of grossly correct abstract plans. An abstract plan must be expanded to the level of action detail required for its execution.

We do not directly address the issue of hierarchical action representation in this dissertation. The topic is first mentioned in chapter 7, where suggestions for future research are made.

Procedural nets allow least commitment plan construction only because the order on actions in a net is partial. This means that a net can contain two (or more) unordered actions, the understanding being that the actions are sufficiently causally independent to permit parallel execution. As discussed above, this is not a feature unique to the procedural net, since state-space plan representations can represent un-ordered actions if defined appropriately. As a matter of historical fact however, the procedural net was the first plan structure which permitted the representation of un-ordered actions.

NOAH performed very little search, and had no backtracking facilities for un-doing bad plan construction decisions. The issue in the NOAH system was one of plan representation, and only basic facilities were included for plan construction. NOAH never made a mistake in constructing the simple plans it produced. A small class of potential plan faults could be dealt with by *critics*. A critic was provided for each plan construction problem that it was necessary to deal with. As in Hacker and Interplan, each critic in NOAH made reference to a plan's underlying teleological structure when suggesting plan modifications. NOAH used a Table of Multiple Effects (TOME) to record the effects of each action added to a plan so that conflicting action additions could be handled. Sacerdoti (1975) explicitly acknowledges the influence of Interplan's Ticklist structures when explaining the function of NOAH's TOME.

System: NOAH.
Plan structure: action-ordering, with limited teleology (TOME).
Parallel/Serial: parallel.
Search space: partial plans, but extremely limited search space.
Search control technique: none.

3.4.9. Nonlin.

Nonlin (Tate, 1977) was designed to correct certain problems found in the NOAH prototype planner. Nonlin was able to search the space of partial plans, while NOAH could not search at all. Nonlin was thus the first planning system to employ a parallel action-ordering plan representation

that was also able to search the space of alternative plans for a solution.

The search heuristic used by Nonlin for plan selection is called *one-then-best*. This heuristic calls for focusing on the choice currently being made and trying to select one of the local alternatives which seems most promising. If a failure occurs, the entire set of alternatives which has been generated is reconsidered to select the re-focusing point. See Tate (1976, 1985) for more on how this works.

Another contribution made by Nonlin came in the form of a more declarative Task Formalism (TF) for describing action-ordering operators to a planner. The operators in NOAH were represented as SOUP (Semantics Of User's Problem) code, and were extremely difficult to separate from the internals of the planner itself. In contrast, Nonlin's TF allowed a person writing domain operators to specify actions, preconditions, and ordering relations in a simple, convenient formalism. TF also introduced the notion of *typed preconditions*, now found in many current planning systems. This precondition-typing mechanism permitted a user to specify a plan's teleological structure very simply and graphically. Tate (1984b) calls this information *Goal Structure*, and the representation for plans developed in chapter 4 includes such information as a fundamental component.

Nonlin implemented procedures for using information regarding activity duration and activity start and finish times. This information was attached to each action within a plan, and used during planning to ensure that the plan being generated satisfied given overall metric temporal constraints. The system was one of the first to be applied to large, realistic domains. Nonlin was used to generate Turbine overhaul plans for the South of Scotland Electricity Board, and also produced house-building plans of reasonable complexity.

System: Nonlin.
Plan structure: action-ordering, with explicit teleology (goal structure).
Parallel/Serial: parallel.
Search space: partial plans.
Search control technique: one-then-best search.

3.4.10. Deviser.

Deviser (Vere, 1981, 1985) built on Nonlin, and extended it in many ways. Most notable is Deviser's ability to specify *time windows* as well as *durations* for actions and goals. An action start-time window is a triple of the form *(Est, Ideal, Lst)*, where *Est* is the action's earliest start time, *Ideal* is its ideal start time, and *Lst* is its latest possible start time. A duration for an action is simply an account of how long the action will take to perform; and a duration for a goal says how long the goal condition must be maintained. Deviser includes the time-window mangement routines required to manipulate a network consisting of actions and goals containing these temporal attributes.

Deviser breaks up "activities" into three sorts: events, which are triggered by environmental circumstance; inferences, performable by the planner; and motor actions, which the actor in the plan can carry out. An inference in Deviser is a special sort of activity in that the validity of the assertions it produces depends on the continuing truth of its preconditions. Deviser provides the required machinery to ensure that the preconditions of all inferences remain true.

The ontology of action developed in chapter 2 extends and rationalises the activity types used in Deviser. Deviser's representation still suffers from the problems discussed in chapter 2: it uses the STRIPS-convention of representing actions by precondition-lists, delete-lists, and add-lists. Although Vere has re-named the lists, the technique employed by Deviser is essentially the same as that used by STRIPS and all planning systems since. Thus Deviser still suffers from the criticisms of chapter 2 regarding the representation of actions by STRIPS-like operators.

Events are considered by Deviser to be spontaneous world changes. For instance, if the actor in a plan opens a valve on a tank containing water, an event will occur: the water in the tank, acted on by gravity, will drain from the tank. This sort of event is dealt with in Deviser by linking the specification of the event to an action which causes the holding of its preconditions. This link is achieved by the use of special *CONSECUTIVE preconditions on the event, made true by the preceding action. The representation for plans developed here includes this ability to describe external events, but in a more principled way than done by Deviser.

It is also possible in Deviser to describe scheduled external events. This is an issue we do not directly address. Since the formalism developed here makes no explicit mention of time windows or durations, such events are impossible to describe.

The teleology in Deviser's plans is only implicit. Deviser's representation does not *highlight* plan teleology, and such information only really appears in diagrams of the plan structures. The representation developed here uses teleological information as one of the basic ways of "sticking" actions together to make a plan.

Vere uses information on "function literals" to improve efficiency. Thus Deviser might contain information about the predicate **colour**, which states that no two literals *colour(Obj1,C1)* and *colour(Obj2,C2)* may co-exist in the database if *Obj1* = *Obj2*, unless *C1* = *C2*. We use such information, but in a more general form. The information comes in the form of *domain constraints*, which describe incompatibilities within any given domain state. In the case of the *colour* predicate, Vere's formulation and ours look similar. We would issue a domain constraint of the form {*colour(O,C1),colour(O,C2)*}, which is interpreted to mean that each object has only one colour. For this example, the techniques seem equivalent. In general however, this is not true. This will become clear when domain constraints are formalised in chapter 4.

New work by Vere (1985) concentrates on techniques for attempting goals in different orders. His work makes it clear that parallel plans do not address the linearity assumption, and that a parallel planner must still deal with the order in which it pursues its goals.

The Deviser system is also noteworthy by its application domain: it has been used to generate spacecraft command sequences for the NASA Voyager mission.

System: Deviser.
Plan structure: action-ordering.
Parallel/Serial: parallel, with goal/activity windows and durations.
Search space: partial plans.
Search control technique: goal splicing.

3.4.11. SIPE.

As in the case of Deviser, SIPE (Wilkins, 1984) is on the line of development rooted at NOAH and running through Nonlin. SIPE extends the capabilities of Nonlin in various ways. SIPE's major contributions are in the areas of resource management reasoning, the provision of inferential operators, and user interaction. This latter feature is facilitated by the development of a declarative and descriptive task net formalism, similar to Nonlin's TF. Like TF, Wilkin's net description formalism allows a user to easily give a limited account of a plan's teleology and condition types. Wilkin's calls this teleological information *Plan Rationale*. The main goals of SIPE are easily understood from the derivation of its title: System for Interactive Planning and Execution monitoring.

A feature that SIPE offers and that isn't found in any planning system yet discussed is the ability to postpone commitment on the selection of objects. SIPE does this through constraint satisfaction reasoning. Such reasoning calls for narrowing down the set of objects which satisfy given requirements, until all the requirements for the desired object are known. This can be contrasted with the standard technique of selecting an object arbitrarily, and re-making the choice when the selected object is found to be unsatisfactory.

SIPE also includes some simple heuristics for reasoning about resources. The heuristics are used primarily to order actions in a developing plan. The ordering is performed on the basis of resource analysis: if one action requests a resource, and does not return it, and another (parallel) action requests the same resource, and *does* return it, SIPE will order the non-consuming action before the consuming one. Not all interaction problems fit this resource reasoning framework, but those that do are dispensed with quickly and efficiently by SIPE. Unfortunately, SIPE can overconstrain the search space (by not considering interleavings within certain action clusters), and may thus not always find a valid plan.

Inferential operators are also provided. Similar to Deviser, SIPE allows a user to write inferential operators which are used to derive action outcomes not explicitly stated in action descriptions. An example given by Wilkins (1984) uses an inferential operator to deduce some

postconditions of a blocks-world move operator. But as noted by Wilkins when presenting the example, SIPE's deductive operators really provide only syntactic shorthand, and the result at the end of plan construction is essentially the same as would be achieved by STRIPS-like operators. The inferential operators in the formalism developed in this dissertation *are* fundamentally different from STRIPS operators.

System: SIPE.
Plan structure: action-ordering.
Parallel/Serial: parallel.
Search space: partial plans.
Search control technique: resource reasoning, also human aided.

3.4.12. Molgen.

Molgen (Stefik, 1981) is notable not for its plan representation, but rather, for its ability to perform object selection using least-commitment techniques. The major phases of Molgen's approach call for constraint formulation, constraint propagation, and constraint satisfaction. Molgen is interesting as the first planner to call attention to this method of object selection. The presentation of these ideas in Molgen preceded their application in other planners, such as SIPE and O-Plan (Currie and Tate, 1985).

The representation of time used by Molgen is a basic state-space model. As far as describing time and change go, Molgen is not very interesting. One of the areas for further research suggested by Stefik himself is the integration of least-commitment object selection with least-commitment action ordering. To this author's knowledge, the only work attempting to do this is the O-Plan project of Currie and Tate (1985).

System: Molgen.
Plan structure: state-space.
Parallel/Serial: serial.
Search space: partial plans.
Search control technique: least commitment object selection.

3.4.13. Georgeff's system.

While not technically a planning system, Georgeff's system (1984, 1985) is interesting because of its rich and expressive representation for plans. Georgeff uses a state-space (basically, a finite-state automata) formalism. He has embellished the model to enable description of iteration with termination and conditional execution. However, issues of representing a plan's underlying teleological structure have not been addressed.

Georgeff's plans can also be used to represent least-commitment action-ordering. These plans allow arcs to be labelled with actions which must all be executed to make a state-transition. As will become clear in chapter 4, the projection structures that we define for plan net analysis are similar to what Georgeff presents. However, we derive our state space plan accounts from an underlying causal description of the plan of action; for Georgeff, the state-space account *is* the plan.

Georgeff does not provide for the description of inferences, or true sensory actions. It does seem possible to extend his model, and future work may well do this. As the theory stands, it cannot deal with any reason maintenance or belief consistency maintenance problems, as discussed initially in chapter 2, and explored in more depth in chapter 6.

System: untitled.
Plan structure: state-space.
Parallel/Serial: parallel.
Search space: nil (does not construct plans).
Search control technique: nil.

3.4.14. O-Plan.

O-Plan (Currie and Tate, 1985) is a planning system and overall architecture designed through experience gained with previous major planners. O-Plan stands for Open Planning Architecture, and is designed to exploit ideas about blackboard control architectures (Hayes-Roth, 1979), least commitment action-ordering (Sacerdoti, 1975), least commitment object selection (Stefik, 1981), goal structure (Tate, 1977), and ideas on attaching temporal window information to actions in a plan (Vere, 1981). With regard to this last point, it is interesting to note that much of the metric

temporal reasoning installed in the O-Plan architecture is borrowed from standard Operational Research network theory. Because of this, O-Plan promises to be useful as a real planning system, and more than a laboratory toy. The architecture is designed to be sufficiently flexible so as to act as a framework for the exploration of a variety of planning ideas, new and old.

The current version of the O-Plan system is focusing on dealing with realistic time and resource modelling, extending the basic window manipulation algorithms of Deviser, reasoning about consumable resources, and providing a limited reasoning ability for re-usable resources.

System: O-Plan.
Plan structure: action-ordering, with explicit teleology.
Parallel/Serial: parallel.
Search space: partial Plans.
Search control technique: least commitment and opportunistic.

3.5. What do most current planners do?

3.5.1. Overview.

Using the comments of the last section, we now discuss features common to the plan representations used by major modern planning systems. Both desirable and undesirable features are discussed, in order to motivate the representation for plans to be developed in chapter 4. The section begins with a breadth-first point-form itemisation of desirable and undesirable features, and then delves more deeply into two major issues: the difficulty of analysing action-order plans in which the order is partial, and the inability of such representations to describe iterative behaviour. Those aspects of the plan representation developed in chapter 4 pertaining to plan teleology are motivated with a two simple examples: opening a door, and hammering a nail.

3.5.2. Common features: desirable and undesirable.

It is possible to draw-out features common to many current planning systems and their plan representations. In particular, it seems that a significant number of the more recent systems discussed above share the following *desirable* features.

(1) Teleological information is included in the description of a plan.

(2) Action-ordering plan representations are used due to compactness, and also because they facilitate user interaction. With regard to point 1, action-ordering representations also seem to make it easy to attach teleological information.

(3) Plans are constructed by searching through a space of partial plans. An analysis of plan teleology often guides the search.

(4) There has been a general trend towards attempting to provide a greater range of action and condition "types" within the same operator description formalism. This sort of specialisation allows the system to more carefully consider how to plan to satisfy goals. Some planners allow the description of simple inferential operators, and operators that describe external events.

(5) There is usually a perspicuous formalism for the description of operators. The formalism is often distinct from the planning system that interprets it, unlike early systems in which the operators and planning systems were made from much the same "stuff".

The representation for plans we develop includes all these features. That is, our plans are defined in terms of an underlying teleological structure, and the teleology is used to define an order on actions. Formal plan analysis is possible, and the analysis procedures make use of a plan's underlying teleology. The results of any given analysis can guide a planner's search. We can represent the range of actions suggested by the examples of chapter 2. We thus expand on the action types of previous systems, and provide a *rationale* for their derivation. Some of the formalism we use for expressing plans comes from Net Theory (Brauer, 1979). Thus, the notation is clearly independent from any particular computer program implementation. Basic constructs in set theory are used to express the structure and function of a plan. This can be contrasted with even the more progressive planners discussed above, which still use "computer-ish" languages to describe plans and operators. The formalism we define makes no commitment to any particular syntax for an implementation.

In addition to common desirable features, it is possible to distinguish a number of *undesirable* features common to the systems discussed.

(1) Teleology is often only "tacked on"; it has never been used to define the *structure* of a plan; rather, it has only ever been drawn on-top of an existing structure.

(2) The order on actions in the action-ordering representations discussed has always been partial. This means that it is impossible to represent iterative behaviour in the structure of one of these nets. This point is very important, and is discussed at length below.

(3) Using current action-ordering representations, it is impossible to naturally represent contingencies; that is, "OR"-type constructs. Such constructs are important in realistic plans.

(4) There has been an unprincipled introduction of different types of action into plan description languages.

(5) All the systems discussed continue to use the STRIPS action representation scheme: actions are characterised by some variation on precondition-lists, add-lists, and delete-lists. This is limited, and poses problems as argued in chapter 2.

(6) There are few principled and implementation independent plan analysis tools. This causes problems for those embarking on the construction of a planner, since it is often unclear how to use the plan decision procedures of other planning systems. We return to this problem of plan analysis below.

(7) All of the planners discussed above are unable to plan to sense, or plan to interact with external events. Some planners can *represent* inferences, but is unclear if they can *plan* to infer. The papers describing the systems certainly don't present any examples of such planning. Some scenarios where such behaviours are necessary have been given in chapter 2. It would be useful and interesting to have a planning system able to generate plans for these behaviours.

The representation for plans developed in this work improves on existing representations on all of these points. In particular, we use teleological information in a basic way to define the very structure of a plan, instead of tacking the information on to a predefined plan structure. This allows us to define a richer ordering than provided by procedural net derivatives, in which the order on actions must be partial. The plans defined here permit the description of iterative behaviour, while retaining an action-ordering plan representation. The plans are also able to represent disjunction of action, or action contingency. Once again, this results from using teleological information to define the structure of a plan. In chapter 2, we discussed our basic ontology of action: while not resting on rock-like philosophical foundations, our ontology seems stronger than those offered by other representations. Chapter 2 also discusses why (and basically how) we reject the STRIPS-form operator specification. The representation formulated in chapter 4 is quite different from the classic STRIPS-form operators. One obvious result of our formulation is the ability to write sensory operators -- operators which describe the results of a sensory action on an agent's set of beliefs. To analyse a plan, we offer the definition of its *projection* -- in essence, a forward simulation of what can happen, given a plan and an initial state. This definition can form the basis for many different sorts of plan analysis.

A point not raised above, and one not directly addressed by this work, is the issue of hierarchy of action. There are various versions of this idea around, and it seems as though it should be possible to use existing tools from Net Theory to analyse and refine what we mean by "hierarchy" and "action" (see Czaja, 1983, for some inspiration).

So we can see that current systems primarily use action-ordering (procedural net derived) plan representations, and search through a space of partial plans. As discussed briefly above, such representations are hard to analyse, and cannot represent iterative behaviour. As we claim that the plan representation developed here does not suffer from these problems, the issues bear further discussion. The next sub-section addresses the analysis of partially ordered action-order plan representations. The discussion is used to argue the point that such representations are unable to describe iteration.

3.5.3. On analysing action-ordering plans.

The difficulty of analysing an action-ordering plan has been recognised by workers in the field. For instance, Charniak and McDermott (1985) note that:

> [The assertions which are true after the execution of a task] ...depend in general on the state of the world when the task is performed, and this state depends in turn on the ordering of tasks, which is only partially known. For example, consider the simple action (move x y), "Move block x to the top of block y." One effect of this action is that the block where *x* originally was now has nothing on it. But which block this is will depend on how the move action is ordered with respect to other planned actions.
>
> (Charniak and McDermott, 1985, p.504).

This kind of uncertainty with respect to what is or is not the case can cause problems in plan construction. In particular, there can be problems with performing the following plan construction operations.

(1) Finding outstanding goals.

(2) Determining goal violations.

(3) Finding variable bindings for operators.

(4) Guaranteeing plan executability.

All these problems arise from uncertainty about the holding of assertions within a plan. The difficulty with performing these operations on a state-space plan is greatly reduced, since determining the status of an assertion is trivial. Given an assertion and a state, one need only check whether or not the assertion is in (or perhaps, is logically entailed by) the state. The lookup operation is very simple. This is manifestly not the case with partially ordered action-ordering plan. Given a partially ordered network of actions and an assertion, a procedure for determining whether or not the assertion holds *may* have to consider all possible interleavings of actions to correctly determine the assertion's status.

Chapman's (1985) *modal truth criterion* provides the necessary and sufficient conditions for an assertion to be true at a point in a procedural-net-like partial order of actions. As well as presenting the criterion, he also discusses the conditions under which it fails. It turns out that for representations which allow for synergy between actions, or for ones that permit "deductive"

operators, the modal truth criterion is inadequate. This is because a full interleaving analysis is often required to determine the status of an assertion at a point in the partial order; the set of all possible interleavings being used to uniquely determine the assertion's status.

In what follows, we assume that the modal truth criterion does not hold. As will be made clear in chapter 4, the criterion does not characterise the sufficient conditions for an assertion to be true at a given point in a plan net, so the assumption is reasonable for current purposes.

The following discussion uses the phrase "procedural net" to refer to the plan representations employed by NOAH (Sacerdoti, 1975), Nonlin (Tate, 1977), Deviser (Vere, 1981, 1985), SIPE (Wilkins, 1984), Tweak (Chapman, 1985), Forbin (Miller, Firby, & Dean, 1985), and PlanX10 (Sridharan, 1982). While each of these systems has its own particular flavour of net, our discussion makes use of only the basic idea of a partial order on a set of actions. Each system makes minor modifications to this basic idea, but each retains the core. Think of the phrase "procedural net" as a parameter, and fill it in with the name of the representation used by any of the planning systems just listed.

One way to answer questions about the holding of assertions in a procedural net is to create a state-space account of it. The problem then becomes that of answering questions about an explicit state-space representation. While this translation may not be computationally cheap, it certainly simplifies the analysis of a procedural net. There are also ways to make the translation from net to state-space graph cheaper, by performing translation on demand as discussed in section 3.2, above (and essentially as characterised by Chapman's modal truth criterion). To simplify the discussion however, we assume that a complete translation is performed when question answering is carried out.

We can produce a state-space account of a procedural net by playing a version of the *pebbling game* (Pippenger, 1980). In our simple version of this game, we place "pebbles" on the nodes of a procedural net when they have been executed. The net starts out pebble-free, and finishes up pebble-laden -- each node must be pebbled; that is, each action must have been executed. Pebble placement is carried out according to the following rule: *A node may be pebbled*

if all of its immediate predecessors are pebbled.

A procedural net is given in figure 3.3, and a state-space account, derived via repeated application of this rule, is given in figure 3.4. In an actual implementation, each action will have associated with it a specification of some changes to be made to a world model. If this specification takes the form of a STRIPS-like add-list and delete-list, then the contents of each state in the derived state-space will be computed in terms of these lists.

While this technique of translating a procedural net into a state-space representation is historically novel, Lansky (1985) has recently proposed to use a similar technique. The project proposed by Lansky includes the development of a planning system which constructs action-ordering plans.

> Each plan constructed by our planner will be described by a partially ordered set of actions... . The planning algorithms must guarantee that, given *any* physical execution of a plan, it will adhere to the rules of the domain and achieve the desired goal.
>
> (Lansky, 1985, p.14)

Lansky proposes to provide this guarantee by creating a state-space account in a manner similar to the one we suggested for the procedural net. It might be thought that analysing all possible interleavings of actions in a partial order will be extremely expensive computationally.

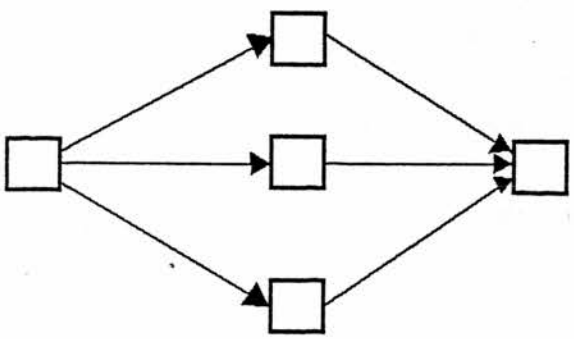


Figure 3.3: A procedural net consisting of 3 parallel actions.

As noted by Lansky:

> One of the problems we will confront in developing our constraint adherence algorithms is how to reason about the potentially explosive number of executions of a given partial order.
> (Lansky, 1985, p.19).

But if we adopt the action-ordering least-commitment plan construction policy (see section 3.2, above), it is possible to reduce the cost of the analysis. It is not necessary to reason about all possible action interleavings if the actions are causally independent; that is, if the actions share no pre- and post-conditions, then they can be applied in any order whatsoever. A successor state can be derived by applying all actions in bulk. Analysis can then continue from the state which results.

It seems possible then, at least in principle, to analyse any given procedural net while retaining an action-ordering least-commitment policy by deriving a state-space structure, and using this structure to do all question-answering. But pebbling can be used for more than this. Above, it was claimed that the procedural net cannot describe iterative behaviour. Pebbling holds a key position in an argument for this claim.

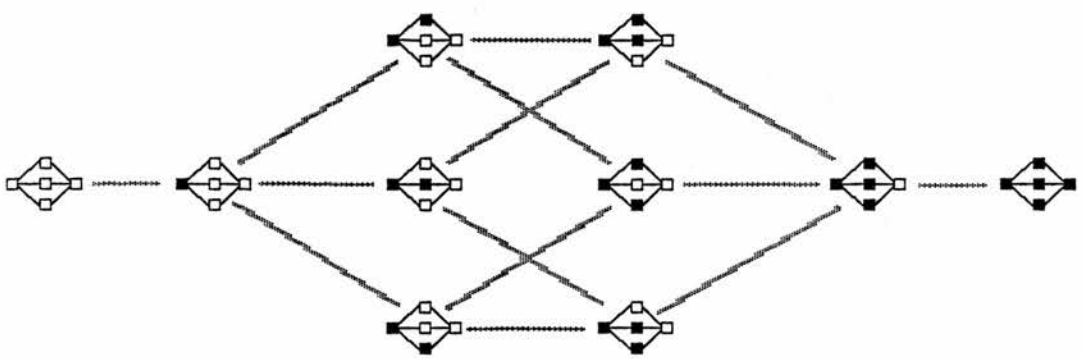


Figure 3.4: The state space of the net in figure 3.3.

Referring back to the procedural net of figure 3.3, and to the pebbling rule, it can be easily seen that all procedural net state-spaces will be loop-free, since the number of pebbles on a net must increase monotonically. There will never be an action which removes a pebble; thus never an action which can produce an earlier state. This means that each action in a procedural net will be executed only once. This is due to the strict "before" interpretation of the procedural net's arcs.

Why is the procedural net unable to describe iterative behaviour? Sacerdoti did actually include a mechanism for dealing with iteration, but it hides the notion of "process" inside a special *replicate* node. This treatment of iteration poses problems. We suggest that by defining an alternative action-ordering representation for plans, it is possible to cleanly describe iterative behaviour, using teleological structure in the basic representation. The key to defining such an action-ordering representation lies in bringing the teleology of a plan into the plan's definition. This is accomplished here by elevating the status of assertions: instead of being *implicit* in the structure of a net, they are made *explicit* in the net's definition.

If ordering relations are defined over actions *and* assertions in a net, then it becomes possible to describe iteration. The procedural net makes no *clear* distinction between assertions and actions. If the distinction is made, and assertions and actions are *both* treated as first class entities, then two *new* sorts of order can be defined: an order on assertions and actions, and an order on actions and assertions. In the following discussion, we refer to the order on assertions and actions as *enable*, and the order on actions and assertions as *cause*. Thus for now, we will say that assertions *enable* the occurrence of actions, and actions *cause* the holding of assertions.

For instance, we might say that the action of opening a door *causes* the door to be open. The fact that the door is open *enables* the action of walking through the door. Certainly, the act of opening the door must come *before* the act of walking through it, but if this is all that is said regarding the two actions, then some information has been omitted. Indeed, we might want to distinguish two sorts of "before" order: one such as that for the door opening example, in which the two actions do have some sort of teleological relationship; and one which is purely agent

"preferential"; that is, one which the agent desires, and not one which the environment enforces. This crude idea is formalised in chapter 4.

Although it might jump the gun a bit, here's an example of how distinguishing between assertions and actions allows one to describe iteration. Imagine two actions: lifting a hammer, and dropping a hammer. There are two corresponding assertions: the hammer is up, and the hammer is down. According to the account given in the last paragraph, we would say that lifting the hammer causes the hammer to be up, and dropping the hammer causes the hammer to be down. The fact that the hammer is down enables lifting it, and the hammer being up enables dropping it. If a plan is defined using these ideas, there would seem to be no problem in describing some behaviour of the form lift, hit, lift, hit, ... Of course, as described, this behaviour could continue forever.

These ideas are made more precise in the next chapter, where the *cause* and *enable* relations are formally defined. This section has only attempted to motivate the formal definitions. When the ideas are made precise, the resulting structure can be seen as an extension of Tate's *Goal Structure* (Tate, 1975b, 1984b), and Wilkin's *Plan Rationale* (Wilkins, 1984a). Not only is this information useful for resolving action ordering problems, but it seems as if it will also be useful for monitoring the execution of plans (as suggested in Tate, 1984a). An understanding of a plan's teleology appears to permit an execution monitoring system to detect problems as soon as they arise, and also allows the repercussions of any given plan failure to be easily determined (Tate, 1984a; Wilkins, 1984b).

3.5.4. Summary.

This section has presented a picture of the plan representations used by the majority of modern planning systems. In general, the plans are action-ordering, with a partial order on actions. Thus, these representations are parallel, in the sense of section 3.2, and are descendents of the procedural net, as pioneered by Sacerdoti (1975). The trend is towards embellishing these procedural net derivatives with richer teleological information. Such information permits better plan construction reasoning and execution monitoring.

We have argued that these plans are in general hard to analyse, and cannot describe iterative behaviour. The technique of Pebbling a net to find its possible states of execution played a key role in these arguments. Pebbling can be used to create a state-space structure for question-answering purposes. This makes question-answering as simple as when using explicit state-space plans. Using pebbling, it is possible to say that no partially ordered network of actions can describe iterative behaviour. This comes from the fact that each net state is created only once. No action in a net can ever be re-executed.

The importance of defining a plan in terms of its underlying teleology was demonstrated with two examples. If instead of tacking teleological information on top of an existing plan structure, we use such information to define a plan, then it becomes possible to describe iterative behaviour. The examples of this section are only intended to motivate the formal presentations of the following chapter.

3.6. Chapter summary and conclusion.

So what has this chapter done?

We began by describing alternatives for *plan representation*. Two orthogonal issues were distinguished. First, whether a plan is represented as a state-space or action-ordering structure, and second, whether the representation provides for the description of parallel or serial plans.

Second, we examined the issue of a planner's *search space*. We distinguished two sorts of search space for a planning system: world states and partial plans. If a planner searches through world states, then it is using its search space navigation procedure to do its plan construction reasoning. If instead, the planner searches a space of partial plans, then it requires two separate procedures: one for navigating the search space, and one for deciding things about plans. The sorts of decisions to be made are about issues like detecting outstanding goals, finding bindings for operators, and guaranteeing plan executability. All these operations depend on what we have called *question-answering*; that is, deciding whether or not a given assertion holds at a specified point within a plan. How a planner does question-answering depends on its plan representation.

For state-space plans this is simple; for action-ordering plans it is sometimes quite hard (basically, it depends on the expressiveness of the underlying operators; see Chapman, 1985).

In addition, a point of terminological confusion was cleared up, or at least avoided: we argued that parallel plans (some say "Nonlinear plans") have nothing to do with the linearity assumption, as embodied in search heuristics such as means-ends analysis.

The comments of the first two sections regarding representation and search were used to review previous planning systems. The review was necessarily incomplete, but an attempt was made to cover all *major* planning systems.

The planning system review was then used to abstract features common to most modern planning system representations. The majority of these systems use representations based on Sacerdoti's (1975) procedural net, and have begun to add teleological information to enable more sophisticated plan construction reasoning. Most of these systems search a space of partial plans.

A technique called *pebbling* was used to argue two related points: first, that analysing arbitrary partial orders of actions can be hard; and second, that procedural nets cannot describe iterative behaviour. The analysis can be made easier if least-commitment action-ordering is used, since the number of action interleavings that must be considered is reduced. We have suggested that plans can be defined which are able to describe iteration if teleological information is used to define the structure of the plan, and not simply laid on top of a predefined plan. Chapter 4 formalises this suggestion.

The plan representation developed in chapter 4 is action-ordering, and can be used to define parallel plans. A planning system which makes use of the representation should search a space of partial plans, and use the teleological plan structure to guide its search. The representation defines a plan essentially in terms of its teleological structure. The plans we define are able to describe iterative behaviour. The definition of plan projection presented in chapter 4 also makes use of net analysis procedures quite similar to pebbling.

Chapter 4: Plan Net Definition.

Everybody calls "clear" those ideas which have the same degree of confusion as his own.

-- Proust

4.1. Chapter Overview.

This chapter explains plans, planning problems, and specifies when a given plan offers a potential solution to a problem. To accomplish this, we define basic nets, operators, plans, domain constraints, the belief consistency maintenance function, and plan net projection. We also present a plan net interpreter, which actually carries out the plan of action specified by any given plan net. To compel and motivate, examples are given throughout. Our examples are taken from the somewhat uninspiring blocks-world domain. While this domain is not stunning in complexity, it is useful for didactic purposes. In the next chapter more complex examples are given, taken from more interesting domains.

As far as formal background goes, only basic knowledge of set theory is assumed. In addition to the standard set-theoretic notation $\cap$, $\cup$, $\in$, $\subset$, $\subseteq$, meaning set intersection, set union, set membership, subset, and improper subset (respectively), we use the following notation.

(1) $-$ indicates set difference, or subtraction; thus $\{a,b,c\} - \{b\} = \{a,c\}$.

(2) $\Pi(P)$ denotes the powerset of the set P; that is, the set of all subsets of P. Thus

$$\Pi(\{a,b,c\}) = \{\{a,b,c\},\{a,b\},\{a,c\},\{b,c\},\{a\},\{b\},\{c\},\{\}\}.$$

(3) Let M be a set, and let $\rho \subseteq M \times M$ be a relation over M. We often abbreviate $(x,y) \in \rho$ as $x \rho y$.

(4) Let M be a set and let $\rho,\sigma \subseteq M \times M$ be two relations over M. We define the transitive closure of a relation as follows.

i) $\rho \bullet \sigma = \{(x,z) \mid \exists\ y \in M: x \rho y \ \&\ y \sigma z\}$

ii) with $\rho^0 = \{(x,x) \mid x \in M\}$ and $\rho^{i+1} = \rho^i \bullet \rho$ $(i = 0,1,2,...)$, let $\rho^* = \bigcup_{i=0}^{i=\infty} \rho^i$

Some of the formalism we use comes from Net Theory (Brauer, 1979). While a few Net Theory constructions are borrowed directly, most are unique to this work. Comment will be made when significant departures are made from standard net theory. Readable introductions to Petri Nets and Net Theory in general can be found in Nielsen and Thiagarajan (1984), Reisig (1985), and Peterson (1977, 1980).

4.2. Structure.

We begin by defining the *terms* of a base language, L. These terms are used below in the definition of the basic formulae from which plan nets are built.

Definition 1. The *terms* of L are of two categories:

(1) A denumerably infinite set of *constants* $Co = \{a_1, a_2, a_3, \cdots\}$. We will sometimes use a to stand for a_1, b to stand for a_2, and c to stand for a_3. We extend this set of constants as required for each domain. There is nothing fundamental in this extension. Each new constant introduced can be considered to stand for some $a_i \in Co$.

(2) A denumerably infinite set of *individual variables* $Va = \{v_1, v_2, v_3, \cdots\}$. We will sometimes use w to stand for v_1, x to stand for v_2, y to stand for v_3, and z to stand for v_4.

Formulae of the language L_B (for Belief Language) are defined using terms from the base language L, and arbitrary *n-place* predicates. Formulae are formed in the obvious way using the predicates and terms. A formula from L_B will be interpreted here as a possible belief. A set of formulae of L_B is to be interpreted as a possible state of belief. If any given formula of L_B is in the agent's *world model*, then the agent might *currently* believe that formula. Other sets of formulae of L_B are only *possible* states of belief. The relationship between the agent and any given potential belief state is determined by the computational relationship between the two. We are only concerned here with states of belief that are possible for the agent in the future. We make this precise later, by defining the projection of a plan into the future, and by defining when any given L_B formula is a justified belief and when it is only an assumption. For now, consider L_B formula to be *possible* (past, present, or future) beliefs.

Definition 2. The *basic expressions* of L_B are of two categories:

(1) The terms of L.

(2) A set of *1-place predicates*, a set of *2-place predicates*, ..., a set of *n-place predicates*.

Definition 3. The *formation rules* of L_B consist of the following:

(1) If σ is an n-place predicate of L_B and $\alpha_1,\alpha_2,\alpha_3,\cdots,\alpha_n$ are terms of L, then $\sigma(\alpha_1,\alpha_2,\alpha_3,\cdots,\alpha_n)$ is a *formula* of L_B.

(2) Nothing else is a formula of L_B.

For a blocks-world example, suppose that L_B contains the set of one-place predicates {*table, block, clear, light, eyes*}, and the set of two-place predicates {*on, weight*}. For this example, we require no predicates of arity greater than two. Let us add to the set of constants *Co* the following: {*open, closed, on, off, a, b, c, d, 1, 2, 3*}. Thus according to the formation rules in definition 3, the following is a subset of L_B: {*on(a,b), eyes(open), block(c), weight(b,2)*}.

We suggest the following interpretations for the basic expressions and formulae just given. The constants *a, b, c, d* denote objects. The sorts of objects considered are blocks and tables. The one-place predicates *block* and *table* apply to objects, and are true if the object argument is a block or table, respectively. The predicate *clear* also applies to objects, and is true if the argument object has enough upper horizontal surface to support a block. The predicate *eyes* applies only to the constants *open* and *closed*. *eyes(open)* is true if the eyes of the agent are open, and *eyes(closed)* is true if the eyes of the agent are closed. (Imagine that there is a lens cap on the video camera that is a component of the agent's visual apparatus; *eyes(closed)* is true if the lens cap is on, and *eyes(open)* is true if the the lens cap is off.) The predicate *light* applies only to the constants *on* and *off*: *light(on)* is true if the light bulb suspended in the blocks-world is illuminated, and *light(off)* is true if the bulb is dark. If the object denoted by *x* is supported by the object denoted by *y*, then *on(x,y)* is true. A formula *weight*(*b*,2) is true if the object denoted by *b* weighs the amount denoted by 2. Valid first arguments for *weight* are objects; valid second

arguments must be drawn from the set {1,2,3}. The constants 1, 2, and 3 denote (respectively) light objects, objects of moderate weight, and extremely heavy objects. These predicates and constants allow us, in conjunction with the variables in *Va*, to construct formulae of L_B which describe various features of a simple blocks-world environment.

Similarly, we now construct the language L_E (Event Language). L_E formulae use terms from the base language L, and predicates of arbitrary arity. Formulae of L_E are built analogously to those of L_B. Each formulae of L_E is intended to denote an action drawn from the ontology developed in chapter 2.

Definition 4. The *basic expressions* of L_E are of two categories:

(1) The terms of L.

(2) A set of *1-place predicates*, a set of *2-place predicates*, ..., a set of *n-place predicates*.

Definition 5. The *formation rules* of L_E consist of the following:

(1) If σ is an n-place predicate of L_E and $\alpha_1,\alpha_2,\alpha_3,\cdots,\alpha_n$ are terms of L, then $\sigma(\alpha_1,\alpha_2,\alpha_3,\cdots,\alpha_n)$ is a *formula* of L_E.

(2) Nothing else is a formula of L_E.

For our blocks-world example, the one-place predicates might be: {*see-light, feel-eyes, block-is-clear, table-is-clear*}; the two-place predicates might be: {*see-on, move-eyes, switch-light*}; and the three-place predicates might be: {*move*}. For this example, we require no predicates of arity greater than three. Using constants in *Co* (as extended above), the following are thus formulae of L_E: {*move(a,b,c),switch-light(off,on)feel-eyes(open)*}. Of course, a great many more formulae can be constructed, most of which make no sense according to the intended interpretation.

We suggest the following interpretation of L_E formulae. The one-place predicate *see-light* applies only to the constants *on* and *off*, and denotes an action of the agent seeing the status of the blocks-world light. *feel-eyes* applies to the constants *open* and *closed*, and denotes the action of the agent sensing whether its eyes are open or closed. The predicates *block-is-clear* and *table-is-clear* apply to blocks and tables, respectively, and denote the action of inferring the clearness of the argument object. For the two-place predicates, we have the following interpretations. *see-on(x,y)* applies to objects, and denotes the action of the agent sensing an *on(x,y)* relationship. *move-eyes* applies to the constants *open* and *closed*, and denotes the action of changing the status of the eyes from open to closed, or from closed to open. *switch-light* applies to the constants *on* and *off*, and denotes the action of switching the light bulb from lit to unlit status, or back. The three-place predicate *move* applies to objects, and denotes the action of moving a block from an initial support to a final support. For this example, we suppose that the agent has a weight sensor in its block movement device (say, a robot arm) such that following the movement of any given block, the agent will hold a belief regarding the block's weight. The weight will be measured in the trinary 1, 2, 3 scheme discussed above. For instance, *move(a,b,c)* denotes the action of moving the block denoted by *a* from the object denoted by *b* (table or block) to the object denoted by *c* (table or block). Following execution of this action, the agent will hold a belief about the weight of the block denoted by *a*.

Using formulae of L_B and L_E, we can now define the basic net structure used to build plans. A net is a 3-tuple consisting of a set of formulae of L_B, a set of formulae of L_E, and a flow relation. These two sets of formulae must be disjoint, and both must be finite in size. The flow relation is a tuple consisting of 4 parts. Each part is a relation on the formulae contained in the sets from L_E and L_B. A net is the basic structure used here, and is employed in the definition of operators and plans below.

Definition 6. A 3-tuple $N = (B,E;F)$ is called a *net* iff

(1) $B \subseteq L_B$ (B is a set of formulae of L_B);

$E \subseteq L_E$ (E is a set of formulae of L_E);

$B \cap E = \emptyset$ (B and E have no intersection);

B and E are finite. (We call $b \in B$ a *b-element*, and $e \in E$ an *e-element*.)

(2) $F = (In0, In1, Out0, Out1)$ is a four-tuple where $In0 \subseteq (B \times E)$, the *enable-out* relation; $In1 \subseteq (B \times E)$, the *enable-in* relation; $Out0 \subseteq (E \times B)$, the *external-results* relation; $Out1 \subseteq (E \times B)$, the *internal-results* relation.

Graphical convention. We represent a b-element by a circle labelled with the b-element, and an e-element as a box labelled with the e-element. The four relations in F are drawn as shown in figure 4.1.

A blocks-world net constructed according to definition 6 appears in figure 4.2. The corresponding graphical presentation is given in figure 4.3. The intended interpretation for the four parts of a net's flow relation is discussed below. In net theory, nets are defined with a flow relation $F = (B \times E) \cup (E \times B)$. We require a richer structure, as will become clear in what follows.

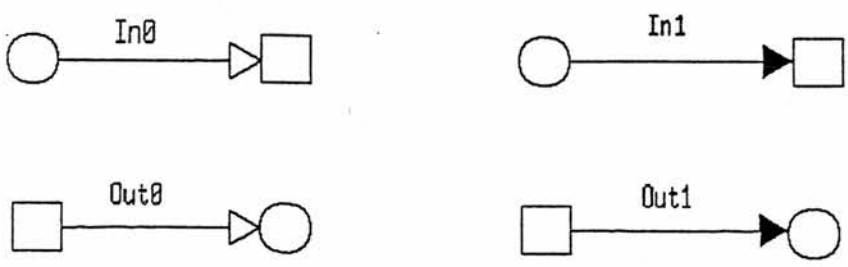


Figure 4.1: Graphical convention for plan net relations.

N = (B,E;F)

B = {light(off), block(x), on(x,y), clear(y), clear(x), on(x,z), weight(x,w)}
E = {move(x,z,y)}

F = (In0,In1,Out0,Out1)

In0 = {(light(off), move(x,z,y))}
In1 = {(block(x), move(x,z,y)),
(clear(y), move(x,z,y)),
(clear(x), move(x,z,y)),
(on(x,z), move(x,z,y))}
Out0 = {(move(x,z,y), on(x,y))}
Out1 = {(move(x,z,y), weight(x,w))}

Figure 4.2: A sample blocks-world net.

Notational convention. Let $N = (B,E;F)$ be a net. We sometimes denote the three components B, E, and F by B_N, E_N, and F_N, respectively. If confusion can be excluded, we also write N for $B \cup E$, and F for $In0 \cup In1 \cup Out0 \cup Out1$. We sometimes denote the four components of F by $In0_N$, $In1_N$, $Out0_N$, and $Out1_N$, respectively.

We often want to make reference to net elements that are connected to each other. Given some particular element of a net, it is useful to be able to refer abstractly to those elements which are directly associated with this element in one of the relations in F. To do this, we introduce the following notation.

Definition 7. Let $N = (B,E;F)$ be a net.

(1) For $x \in N$

$^{\pm}x = \{y \mid y\ F_N\ x\}$ is called the *basic preset* of x;

$x^{\pm} = \{y \mid x\ F_N\ y\}$ is called the *basic postset* of x.

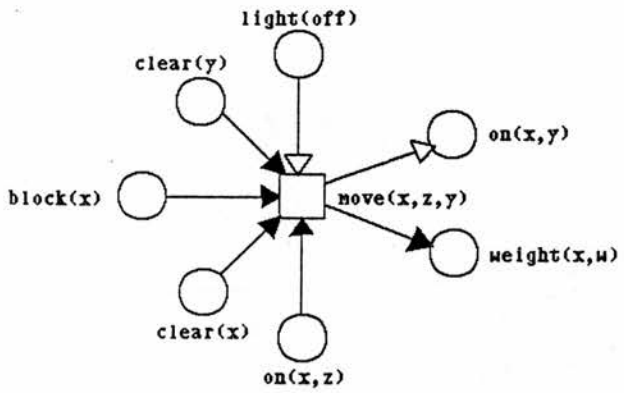


Figure 4.3: The net of figure 4.2, graphically presented.

For $X \subseteq N$, *let* $^{\pm}X = \bigcup_{x \in X} {}^{\pm}x$ and *let* $X^{\pm} = \bigcup_{x \in X} x^{\pm}$

(2) For $x \in E$

$^{-}x = \{y \mid y\ In0_N\ x\}$ is called the *enable-out preset* of x;

$^{+}x = \{y \mid y\ In1_N\ x\}$ is called the *enable-in preset* of x;

$x^{-} = \{y \mid x\ Out0_N\ y\}$ is called the *external-results postset* of x;

$x^{+} = \{y \mid x\ Out1_N\ y\}$ is called the *internal-results postset* of x.

As in definition 7.1 (above), we extend this notation to cover sets $X \subseteq E$.

(3) A pair $(b,e) \in B_N \times E_N$ is called a *self-loop* iff $(b\ F_N\ e)\ \&\ (e\ F_N\ b)$. N is called *pure* iff F_N does not contain any self-loops.

(4) $x \in N$ is called *isolated* iff $^{\pm}x \cup x^{\pm} = \varnothing$.

(5) N is called *simple* iff distinct elements do not have the same basic preset and basic postset, i.e.

$$\forall\ x,y \in N\colon ({}^{\pm}x = {}^{\pm}y\ \&\ x^{\pm} = y^{\pm}) \rightarrow x = y.$$

Using the net of figure 4.2, the following are examples of the notation developed in definition 7.

${}^{\pm}move(x,z,y) = \{light(off),block(x),clear(y),clear(x),on(x,z)\}$

$move(x,z,y)^{\pm} = \{on(x,y),weight(x,w)\}$

$clear(x)^{\pm} = \{move(x,z,y)\}$

${}^{-}move(x,z,y) = \{light(off)\}$

${}^{+}move(x,z,y) = \{block(x),clear(y),clear(x),on(x,z)\}$

$move(x,z,y)^{-} = \{on(x,y)\}$

$move(x,z,y)^{+} = \{weight(x,w)\}$

The net given in figure 4.2 is pure, contains no isolated elements, and is simple.

The four ordering relations on e-elements and b-elements describe for the agent the relationship between any given action and that action's enabling conditions and results. The idea is as follows. An e-element describes an action, and the e-element's enable-in preset describes the environmental conditions under which the action can occur. Clearly, formulas in this preset are not themselves the conditions required for the action's occurrence, but rather reflect the agent's understanding of what the real conditions are. Similarly, an e-element's enable-out preset describes the conditions which must not obtain if the action denoted by the e-element can occur. In the plan net framework, this condition on action enablement is translated into the absence of a formula in a set of formulas. The enable-in preset is used to formalise action enablement by requiring that each formula be present in the enabling set of formulas. Each such set of formulas is viewed as a possible "world model", and the enablement of e-elements is defined in terms of

these possible world models. This will become clear below, when we define e-element concession and occurrence.

An e-element's external-results postset describes the external, or physical, results of the denoted action. Such results include the orientation and location of blocks, the positioning of light switches, and the colour of objects. The internal-results postset of an e-element describes the internal, or mental, results of the denoted action. Internal results are those that are realised *directly* on the agent's mental state. Such results are achieved by actions which impinge on the agent's sensory modalities, such as actions which involve vision, audition, and taction. Actions which might be considered to qualify include looking, listening, and touching.

The relationship between any e-element and b-element captures what the discussion in chapter 3 called "teleological" information. Here, we use the term *teleological* to refer to the *reasons* for some action or belief being in a plan. The ordering relations defined above describe for the agent very precisely the causal connections among actions. These ordering relations allow a formal analysis of which actions can be used to enable which other actions; this is essentially the reasoning that is required of a plan generation system.

Consider again the net of figures 4.2 and 4.3. This net contains only one e-element: *move(x,z,y)*. The e-element denotes the action of moving the block denoted by x from an initial support denoted by z to a final support denoted by y. The basic preset of the e-element describes what formulas must and must not be in the world model if the denoted action is to be enabled. More specifically, the e-element's enable-in preset says that the action denoted can only occur if the formulas *block*(x), *clear*(y), *clear*(x), *on*(x,z) are all in the world model, and if the formula *light*(*off*) is *not* in the world model.

Further, the net determines that the *move* action has two results, specified by the e-element's basic postset. In particular, the move action has the external result that the block denoted by x is on the object denoted by y. This action also has the internal result that the block denoted by x has a weight denoted by w. Thus, this action is "sensory", in that it imparts to the agent a measure of the moved block's weight. It is certainly true that the block would have a weight before the

action occurs; however, the agent may or may not have had any beliefs regarding this weight. After the action occurs, the agent *will* have some sort of belief about the weight of x.

We can now define operators (and plans) as nets which have certain properties. An operator may have *isolated* elements. An operator will be taken to be *pure*, since the definition of the execution of impure nets is difficult. In addition, to be an operator, a net must contain only unique components; that is, it must be *simple*. The definition of an operator marks another point of departure from standard Net Theory. We include the definition of a *before* relation, used to constrain the way in which a net can execute. Here, the before relation is simply introduced as a part of an operator. How this is used is made clear in the definition of problems and their solution, below. Intuitively, the before order is a specification of which events must occur before which other events, if a plan is to run to its intended completion. We often refer to the *before* relation as *execution advice*.

Definition 8. A 2-tuple $O = (N,A)$ is called a *plan net* iff

(1) N is a pure and simple net.

(2) $A \subseteq \Pi(E_N) \times \Pi(E_N)$ is called the *before relation*. This relation must be a strict partial order: irreflexive and transitive, thus asymmetric. We use the terms *before relation* and *execution advice* interchangeably. (In graphical presentations, if $A = \varnothing$, nothing will be shown for the execution advice; otherwise, the pairs in A will be listed with the plan net.)

(3) If $| E | = 1$ then we also say that O is an *operator*.

What is the difference between an operator and a plan? As used here, the terms reflect two perspectives. We will call a net plus advice an *operator* when it is given (perhaps with other operators) to a planner to be used in solving a specified problem. To be useful, such an operator will usually contain individual variables, but this is not always so. An operator might be of some use in solving a problem even if it contains only constants. A net/advice two-tuple will be called a *plan* when it is produced by a planner as a potential solution to a problem. Of course, this plan

may later be used as an operator, if the problem solver has the ability to store and re-use previous solutions. Thus there is no fundamental distinction between operators and plans in this formalism, and the terms will be used as appropriate, depending on context.

The requirement that operators must have only one e-element reflects a preference for simple operator analysis. This requirement can be relaxed, and only affects the amount of analysis required of a plan construction system. With only one e-element, it is obvious what the net can be used for (i.e., what its final postconditions are). With more than one e-element, net analysis is required to find overall "inputs" and "outputs".

According to definition 8, we can construct an operator from the net N which appears in figure 4.2 as $O = (N,\varnothing)$; Thus O is an operator with execution advice that is the empty set. This seems reasonable, since it's hard to imagine requiring ordering advice for the execution of a single action!

A larger blocks-world example is given in figure 4.4. The structure in the figure is a blocks-world plan for building a stack of blocks: block *a* on top, block *b* in the middle, and block *c* on the bottom. This net can be considered to be constructed from two instantiated versions of the net N in figure 4.2. The derivation of the net of figure 4.4 from the net of figure 4.2 is an issue of plan generation, and we defer discussion of this until chapter 6.

4.3. Consistency.

It is not enough to have a simple structural account of a plan of action. We also require some means for describing the behaviours it specifies. In this thesis we use a conventional state-space (see chapter 3) account, which we call a *projection*. In order to define a plan's projection, we must first address the definition of the belief consistency maintenance function, μ. This function will be used in defining a basic state transition, the atomic "step" from which plan projections will be constructed.

The basic idea of the μ function is as follows. Given a set of current beliefs, and a set of new observations (potential beliefs), we must arrive at a peaceful union of the two sets. It is this

```
P = (B,E;F,A)

B = {block(a), block(b), block(c), clear(a), clear(b), clear(c),
     on(a,b), on(a,t), on(b,c), on(b,t), table(t), light(off)
     weight(a,x), weight(b,y)}
E = {move(a,t,b),move(b,t,c)}

F = (In0,In1,Out0,Out1)

In0 = {(light(off), move(a,t,b))
       (light(off), move(b,t,c))}
In1 = {(block(a),  move(a,t,b)),
       (on(a,t),   move(a,t,b)),
       (clear(a),  move(a,t,b)),
       (clear(b),  move(a,t,b)),
       (clear(b),  move(b,t,c)),
       (block(b),  move(b,t,c)),
       (on(b,t),   move(b,t,c)),
       (clear(c),  move(b,t,c))}
Out0 = {(move(a,t,b), on(a,b)),
        (move(b,t,c), on(b,c))}
Out1 = {(move(a,t,b), weight(a,x)),
        (move(b,t,c), weight(b,y))}
Out1 = {}

A = { ({move(b,t,c)}, {move(a,t,b)}) }
```

Figure 4.4: A Blocks-world plan.

union that is to become the new set of current beliefs. Clearly, the new observations and the old beliefs may (indeed, probably will) be incompatible. What is required then, is a *reasoned deletion* of old beliefs, such that the set obtained through belief deletion is compatible with the new observations. In saying this, we have made a simplifying assumption: none of the new beliefs may be among those deleted. This can be thought of as saying that our agent will never disbelieve what it senses. The role of the function μ then, is this: given a set of current beliefs, and a set of new observations, return the set of possible deletion sets from the current beliefs, such that any one of the possible deletion sets, when removed from the current beliefs, makes the union of the current beliefs and new observations consistent. Consistency is defined with respect to a set

of domain constraints. A reasonable place to begin then, is by defining consistency for a set of beliefs. To do this we begin with the notion of a *valuation.*

Definition 9. A function β: $Va \rightarrow Co \cup Va$ is called a *valuation.* We will often find it convenient to apply a valuation to arbitrary terms of L, and not just individual variables. To facilitate this, given any particular valuation β, we define $\hat{\beta}$: $Va \cup Co \rightarrow Va \cup Co$ as follows.

$$\hat{\beta}(v) = v' \textit{ iff } v \in Va \ \& \ \beta(v) = v'$$

$$\hat{\beta}(v) = v \textit{ iff } v \in Va \ \& \ \beta(v) \textit{ is undefined}$$

$$\hat{\beta}(c) = c \textit{ iff } c \in Co$$

This usage induces, canonically, a mapping

$$\hat{\beta}: L_E \cup L_B \rightarrow L_E \cup L_B$$

by

$$\hat{\beta}(\sigma(\alpha_1,\alpha_2, \cdots ,\alpha_n)) = \sigma(\hat{\beta}(\alpha_1),\hat{\beta}(\alpha_2), \cdots ,\hat{\beta}(\alpha_n)).$$

We also expand $\hat{\beta}$ for sets of formulae $T \subseteq L_E \cup L_B$ by $\hat{\beta}(T) = \{\hat{\beta}(t) \mid t \in T\}$.

Definition 10. Let $c,c' \subseteq L_B$, and let $X \subseteq \Pi(L_B)$. c and c' are called *cases*, and X is called a set of *domain constraints.*

(1) We say that c' *can be bound in* c, iff there exists an injective valuation β: $\hat{\beta}(c') \subseteq c$. An injective valuation is one which, for all $x,y \in$ *domain*(β), if $x \neq y$, then $\beta(x) \neq \beta(y)$. Requiring the valuation to be injective ensures that different variables in a domain constraint will be bound to different constants when attempting to detect constraint violations.

(2) We say that c *is consistent with respect to* $X \subseteq \Pi(L_B)$ iff $\forall\ x \in X$: x cannot be bound in c.

Definition 11. Let $c,c' \subseteq L_B$ be cases, and let $X \subseteq \Pi(L_B)$ be a set of domain constraints.

(1) A set $q \subseteq L_B$ is said to *reconcile c with c' (with respect to X)* iff

a) $(c \cup c') - q$ is consistent with respect to X;

b) $c' \subseteq (c \cup c') - q$ (or, equivalently, $q \cap c' = \varnothing$);

c) $\neg\exists\ q' \subseteq L_B$ such that $q' \subset q$ & $(c \cup c') - q'$ is consistent with respect to X.

(2) We define the *consistency maintenance function*, μ: $(\Pi(L_B)\times\Pi(L_B)) \rightarrow \Pi(\Pi(L_B))$, with respect to X, as

$$\mu_X(c,c') = \{q \subseteq L_B \mid q \textit{ reconciles } c \textit{ with } c' \textit{ (with respect to } X)\}.$$

It is quite simple to illustrate the use of this function. If we consider the agent to be situated in the blocks-world, and in possession of a set of beliefs, domain constraints, and sensory system, we can construct the following example.

Imagine that the agent's domain constraints, $X \subseteq \Pi(L_B)$ are the following.

$$\{\{\ on(x_1,y_1),\ clear(y_1),\ block(y_1)\ \},$$
$$\{\ on(x_2,y_2),\ on(x_2,z_2),\ block(x_2)\ \},$$
$$\{\ on(x_3,y_3),\ on(z_3,y_3),\ block(y_3)\ \},$$
$$\{\ block(x_4),\ table(x_4)\ \},$$
$$\{\ on(x_5,y_5),\ on(y_5,x_5)\ \}\}$$

The first of these says that it is impossible for the agent to believe there is any block y_1 which is both clear and under some other object. The second says that the agent should never believe that there is any block x_2 which is on top of two different objects simultaneously. Equivalently, we can view this second constraint as saying that each block must be uniquely supported. The third domain constraint specifies that it is impossible to believe that there is any block y_3 which is under two different blocks. This constraint can be considered to say that each block must uniquely support some other block (if it supports any at all). The fourth constraint says that it is impossible for the agent to believe that any given object x_4 is both a block and a table. The fifth and final constraint specifies that it is impossible to believe that two objects are on each other.

Suppose that the agent is in the state of belief $c \subseteq L_B$, as shown below; i.e., assume that the agent's world model contains only those L_B formulae in c.

c = { *clear(a), clear(t), on(a,b), on(b,t), block(b), block(a), table(t)* }

A picture of a state of the blocks-world that *might* correspond with this state of belief is shown in figure 4.5. Notice that not all relevant features of the blocks-world as we have defined are shown: in particular, we omit any description of the light being on or off, and make no commitment to whether the agent's eyes are open or closed. According to the interpretation desired here, this case describes a state of belief for our agent. Thus, we might say that the agent believes that the object denoted by a is a block, and that this block is clear. The agent also believes that the object denoted by t is a table, and that this table is also clear, and so on. (Below we will be more precise about when a formula is a true belief, and when it is only an assumption; for now, some imprecision can be tolerated.)

Now suppose that the the following observation is made available by the agent's sensory system.

c' = {*clear(b)*}

It is easy to see that the sets c and c' are incompatible: in c, there are the beliefs {*on(a,b), block(b)*}, and in c', the belief *clear(b)*. There is a constraint in X which says that *blocks* are not both *clear* and *under* some object. Thus the union of these two sets would not be consistent with respect to X. An evaluation of the consistency maintenance function in this situation should appear as: $\mu_X(c,c')$ = { {*on(a,b)*}, {*block(b)*} }.

This means that a new state of belief, consistent with respect to X, can be obtained by removing {*on(a,b)*} or {*block(b)*} from c, and performing the union of the result with c'.

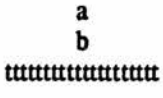


Figure 4.5: A picture of a part of a possible state of the blocks-world.

Depending on which alternative the agent chooses, the following are both possible successor belief states.

{ *clear(b), clear(a), clear(t), block(b), on(b,t), block(a), table(t)* }

{ *clear(b), clear(a), clear(t), on(a,b), on(b,t), block(a), table(t)* }

Notice that this sort of belief consistency maintenance is *not* addressed by Reason Maintenance Systems (RMS). As stated by Doyle (1983, p. 349): "RMS does not maintain consistency of beliefs in any important sense.". Once the *program* using an RMS has decided which beliefs conflict, RMS calculates the follow-on consequences of the decision. μ formalises the calling program's *reasons* for detecting inconsistencies. The relationship between RMS and the representation we develop in this thesis is explored more fully in chapter 7, section 3.

The fact that there will often be a number of choices for which beliefs to reject in order to maintain consistency seems to point out the need for some sort of heuristic. This heuristic could be used to guide the choice as to the most suitable reconciliation set. In the example considered so far, the best choice is rather clear, and points the way towards the formulation of a general blocks-world heuristic. This heuristic might prefer reconciliation sets that do not contain the predicates *block* or *table*. This would reflect the idea that beliefs about the *nature* of objects should be held more tenaciously than beliefs about the *arrangement* of objects.

It is possible to extend this sort of heuristic to other domains. What's required is some measure of belief *volatility;* that is, a measure of the speed with which beliefs can be expected to change. It seems unnecessary to assign absolute numbers to predicates; rather, it should be possible to order the predicates of any given domain to reflect the speed with which beliefs may change.

In the formalisation which follows, we insist that one of the possible reconciliations for an inconsistency be chosen, and the rest ignored. To this end, we require a reconciliation set selection function, ψ: $\Pi(\Pi(L_B)) \rightarrow \Pi(L_B)$. In the definitions below we assume that such a function exists, but do not explicitly define it. ψ must incorporate the heuristic about which reconciliation set looks "best", and is thus entirely domain dependent. It must be specified as

required for each application domain.

4.4. Projection.

We now use the consistency maintenance function in the definition of the simulated execution of an action. We must first define when a given e-element has *concession;* that is, when the event it denotes is believed by the agent to be enabled. In addition, we must define how a case (interpreted as a possible state of belief) changes under the *occurrence* of the action denoted. This definition of e-element occurrence gives us an accessibility relation on belief states for the agent. E-element occurrence will be used as a "state generator", permitting the definition of state-space structures which describe the behaviours sanctioned by a plan.

We have here a rather interesting decision to make, regarding the sort of story we would like to tell about plan projection. There are two obvious projection stories, and they differ as to what they say about how to use an e-element's external-results and internal-results postsets. Recall the motivation behind the introduction of these sets: one set describes the external-results of an action (*a la* Strips), and the other set describes the internal-results of the action.

How should these sets be used in deriving "successor" belief states? Once answer is to use only an e-element's internal-results set as a source of formulae to add to a generated state. This would seem to accurately model the occurrence of the action denoted by the e-element: the action's internal effects (e^+) are added to generate the successor state, and the action's external effects (e^-) have no effect on the belief state of the agent. Given a case c, and an e-element e which has concession in c, we refer to the state transition specified by a next-state generator which uses only e^+ as a *realistic transition.* Plan projections built from realistic transitions will be called realistic projections.

Using only realistic projections to explain net behaviour would be tedious at best. If actions have been included in a plan on the basis of their declared external results postsets, it might prove efficacious for the agent to simply *assume* that the actions will actually function as the corresponding e-element specifies. To do this, the agent could use the external-results set of an e-

element to generate successor belief states. In this way, the e-element would be giving the agent sufficient cause to *assume* given formulas. Given a case c, and an e-element e which can occur in c, we refer to the state transition specified by a next-state generator which uses $e^{\pm}$ as an *assumptive transition*. Plan projections built from assumptive transitions will be called assumptive projections.

So which is the more appropriate execution story to tell? In general, it will depend on the nature of the agent's environment and plan execution system. If the external results of actions are likely to be sensed without intentional action towards that end, then an assumptive projection is adequate. If a domain is sufficiently extended, such that the external effects of an action often occur beyond the agent's immediate sensory range, then it would be more appropriate to employ a realistic projection. However in general, it seems unnecessary to require that each relevant external action effect be monitored by the inclusion of explicit sensory actions in a plan.

It is important to realise that the particular execution story told does not affect the agent's ability to construct sensory plans. The internal and external results of an action are distinguished to the agent by e^{+} and e^{-}, and it is this distinction that allows the agent to index the appropriate "sensory" and "motor" actions, as required. This indexing is performed in terms of the agent's goals, so operators are retrievable for the appropriate result.

If we are to use an assumptive projection, we must provide an adequate definition of *assumption*, such that an assumptive projection is one way of producing assumptions. This isn't to say that there won't be other ways of producing assumptions: what is required is a definition of assumption such that that's what an assumptive projection produces. This sounds almost trivial, but it is important to define assumptions generally, so that the definition applies to situations other than that which arise when doing assumptive projections. Later on, we present a definition of assumption which does this.

We now define both a realistic and assumptive transition. However, in what follows, we will focus primarily on the assumptive transition, and the plan projection structures which are constructed from it.

Definition 12. Let $P = (B,E;F,A)$ be a plan, let $c \subseteq L_B$ be a case, let $X \subseteq \Pi(L_B)$ be a set of domain constraints, and let ψ: $\Pi(\Pi(L_B)) \rightarrow \Pi(L_B)$ be a reconciliation set selection function.

(1) Let $e \in E$. We say that *e has concession in c* (is *c-enabled*) iff

$$^{+}e \subseteq c \;\; \& \;\; \forall \textit{ valuations } \beta\text{: } \hat{\beta}(^{-}e) \cap c = \varnothing.$$

(2) Let $e \in E$, and let e be c-enabled. The *realistic follower case to c under e (respecting X)* is c', given by

$$c' = (c \cup e^{+}) - \psi(\mu_X(c,e^{+})).$$

This is written as $c \xrightarrow[e]{X} c'$, with a particular ψ to be understood from context.

(3) Let $e \in E$, and let e be c-enabled. The *assumptive follower case to c under e (respecting X)* is c', given by

$$c' = (c \cup e^{\pm}) - \psi(\mu_X(c,e^{\pm})).$$

This is written as $c \overset{X}{\underset{e}{\Rightarrow}} c'$, with a particular ψ to be understood from context.

Graphical convention. When drawing a case $c \subseteq L_B$, in terms of a plan $P = (B,E;F,A)$, we place a dot (a *token*) inside each and only those circles labelled with formulae in B which are also in c.

Definition 12.1 expresses the conditions under which any given e-element is allowed to occur (has concession, or is enabled). We interpret the enable-out preset of any e-element as containing those b-elements (formulae from L_B) that describe the conditions that must not hold if the action referenced by the e-element is to be enabled. This is formalised in definition 12.1 by insisting that a condition on e-element concession is $\forall\beta$: $\beta(^{-}e) \cap c = \varnothing$; that is, by insisting that the e-element's enable-out preset cannot *possibly* have any intersection with the given case. Similarly, the enable-in preset is understood to contain formulae that describe those conditions which must hold if the action referenced by the e-element is to be enabled. This appears as the requirement $^{+}e \subseteq c$. This condition on concession ensures that the e-element's enable-in preset is

definitely contained within the given case.

Notice that neither definition of e-element occurrence allows e-elements to directly remove b-elements from the case in which they occur. This is interpreted as saying that no action *directly* causes the agent to stop believing anything. B-elements are only removed through the consistency maintenance function μ, in conjunction with the domain specific reconciliation set selector ψ. What is suggested for removal is a function of the enabling case, and the appropriate postset of the e-element that occurs. The only way a belief is ever removed from the set of current beliefs is if the consistency maintenance function requires it. This means that old beliefs are only removed when they can be shown to be inconsistent with new ones. This is more general than having each e-element commit itself to specifying a "delete-list". What is deleted is quite literally a function of what is currently the case and what is being added to the current case.

It is important to understand that at this point we are still unable to discuss event executancy, as first introduced in chapter 2. However, it is not necessary to have executancy defined for creating a plan's projection. Projection does not need to take event executancy into account, since a plan's projection is a state-space description of the states of belief the plan might give rise to, and creating this description only requires an understanding of the *possibility* of event occurrence. A projection is only a description of what could happen, and does not depend in any way on executancy of action. As far as plan projection goes, it doesn't matter whether the agent or environment has executancy.

For an example of e-element concession and occurrence, consider the plan net shown in figures 4.6 and 4.7, and assume that we still have the blocks-world domain constraints first presented above. We will refer to these domain constraints as X. Let $c \subseteq L_B$ be {*weight(a,1), block(a), on(a,t), clear(a), clear(b), block(b)*}. There is only one e-element in the net, and it has concession in c: *move(a,t,b)*. The realistic follower case to c under *move(a,t,b)* (respecting X) is {*weight(a,x), weight(a,1), block(a), on(a,t), clear(a), clear(b), block(b)*}. This can be explained as follows. Since a realistic follower case is derived by adding the internal-results postset of the e-element, for this example we need only add the set { *weight(a, x)* }. The addition of this single

formula does not violate any constraints, so the realistic successor to c is essentially c itself, with one new formula. What this means is that following the occurrence of the actual *move* action, the agent will have a belief regarding the weight of the object denoted by a. This may or may not be the same as the agent's current belief that this object is of weight 1. It is the responsibility of the execution system to actually throw out the belief *weight(a,1)*, should it prove inconsistent with a new actual weight determined by the performance of the move action.

We can also analyse e-element occurrence in terms of an assumptive transition. Ignoring for a moment the reconciliation set selector function ψ, all possible assumptive follower cases of c under *move*(a,t,b) (respecting X) are

{{ *weight(a,x), weight(a,1), on(a,b), on(a,t), clear(a), block(b)* },

{ *weight(a,x), weight(a,1), on(a,b), block(a), clear(a), block(b)* },

{ *weight(a,x), weight(a,1), on(a,b), on(a,t), clear(a), clear(b)* },

{ *weight(a,x), weight(a,1), on(a,b), block(a), clear(a), clear(b)* }}

These alternatives arise from the choices for reconciliation sets returned by the consistency maintenance function. In this example, μ would return the following set of reconciliation sets.

{{ *clear(b), block(a)* },

{ *clear(b), on(a,t)* },

{ *block(b), block(a)* },

{ *block(b), on(a,t)* }}

Each possible successor is obtained by removing one of these sets from c. It seems that one of the deletion sets is more reasonable than the others: { *clear(b), on(a,t)* }. Each of the other possible deletions contains the predicate *block*. Now while it seems highly unlikely that the objects denoted by a or b have ceased to be blocks, the agent may indeed have incorrect beliefs about the nature of the objects. If the source of the observation *on*(a,b) (say a vision system) isn't 100% reliable, it is possible that parts of the environment have been incorrectly sensed in the past. In particular, it is possible that the object denoted by b was originally (and mistakenly) believed to be a block. This object might in fact be a table, as later sensing will perhaps determine. So while

it is not very likely that the objects denoted by a or b have physically changed, the agent may hold incorrect beliefs about the nature of the objects, and the results of subsequent sensing must be able to correct these erroneous beliefs.

Once we use the reconciliation set selection function ψ to choose from among all the possibilities, we are left with a single assumptive follower to c under $move(a,t,b)$ respecting X. We write this as $c \underset{move(a,t,b)}{\overset{X}{\Rightarrow}} c'$, where

$$c' = \{ weight(a,x), weight(a,1), on(a,b), block(a), clear(a), block(b) \}.$$

This ψ incorporates the heuristic suggested above, which prefers reconciliation sets that do not contain the predicates *block* or *table*. In defining a plan's projection, we use only the reconciliation set returned by ψ to construct successors under e-element occurrence.

There is one more thing that must be done to allow the definition of a plan's projection to proceed. We must define what it means to apply a *set* of e-elements to a case. So far, we have only defined the occurrence of *single* e-elements. This is too simple, since actions need not

---

P = (B,E;F,A)

B = { light(off), block(a), on(a,b), clear(b), clear(a), on(a,t), weight(a,x)}
E = { move(a,t,b) }

F = (In0, In1, Out0, Out1)

In0 = {(light(off), move(a,t,b))}
In1 = {(block(a), move(a,t,b)),
(clear(b), move(a,t,b)),
(clear(a), move(a,t,b)),
(on(a,t), move(a,t,b))}
Out0 = {(move(a,t,b), on(a,b))}
Out1 = { (move(a,t,b), weight(a,x)) }

A = ∅

Figure 4.6: A blocks-world plan net.

---

always occur serially. It may be possible for some number of actions to occur at the same time. Even more important is that by defining the occurrence of a set of e-elements, we achieve the ability to perform least-commitment action-ordering plan construction. As discussed in chapter 3, this means that if some given actions are causally independent they can be applied as a set, and reasoning can continue from the resulting case. To define this more carefully, we begin by specifying when a set of e-elements is *detached;* that is, when the actions they denote are sufficiently causally independent to permit parallel execution. Given a suitable definition of detachment, it becomes possible to define parallel e-element occurrence.

Definition 13. Let $P = (B,E;F,A)$ be a plan, let $X \subseteq \Pi(L_B)$ be a set of domain constraints, and let c, c', $c'' \subseteq L_B$ be cases.

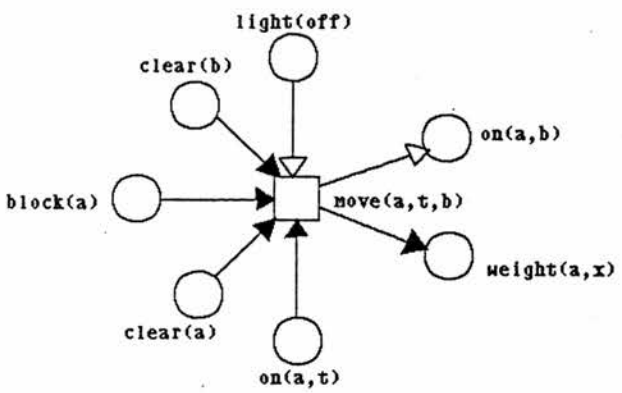


Figure 4.7: The plan net of figure 4.6, graphically presented.

(1) Let $e_1,e_2 \in E$ be c-enabled. We say that e_1 and e_2 are *causally independent in* c iff

$c \overset{X}{\underset{e_1}{\Rightarrow}} c'$ with e_2 being c'-enabled, and $c \overset{X}{\underset{e_2}{\Rightarrow}} c''$ with e_1 being c''-enabled.

(2) A set of events $G \subseteq E$ is called *detached in* c iff

$$\forall e \in G\text{: } e \text{ is } c\text{-enabled } \&$$

$$\forall e_1,e_2 \in G,\ e_1 \neq e_2\text{: } e_1 \text{ and } e_2 \text{ are causally independent in } c.$$

(3) Let G be detached in c. G is called an *assumptive step from c to c' (respecting X)*, written

$c \overset{X}{\underset{G}{\Rightarrow}} c'$, where c' is given by

$$c' = (c \cup G^{+}) - \psi(\mu_X(c,G^{+})).$$

(With a particular ψ understood from context.)

Notice that if G contains only one element, i.e., if $G = \{e\}$, then $c \overset{X}{\underset{G}{\Rightarrow}} c'$ is equivalent to $c \overset{X}{\underset{e}{\Rightarrow}} c'$.

(See Appendix C for a proof of the order-independence of e-element application in a detached set of e-elements.)

It is possible to define an exactly analogous *realistic step*, simply by replacing the assumptive transition specification with a realistic one. However, we define only the assumptive step, since it is generally more useful.

To enable simple state-space reasoning over a plan net, we must define the projection of a plan as a directed graph structure. The nodes of any projection graph are cases, that is, sets of L_B formulae, and the arcs are labelled with sets of e-elements from the plan net being projected. Thus, if an initial node of the projection of a plan is a subset of the current world model, the nodes reachable from that node in the projection describe possible future belief states for the agent.

Having a plan net's projection be a state-space structure greatly simplifies the reasoning required for plan net construction, as was argued in chapter 3. Allowing the arcs to be labelled with *sets* of e-elements further simplifies plan construction reasoning, since all possible interleavings of e-element occurrence need not be considered. If a set of e-elements is detached,

they are causally independent and can execute in any order.

A state-space structure is easily represented as a graph. To this end, we now define arc labelled, oriented graphs.

Definition 14. A tuple $G = (H,P)$ is called an *(arc labelled, oriented) graph over L* iff H and L are sets such that $P \subseteq (H \times L \times H)$. The elements of H, L, and P are called *nodes*, *arc labels*, and *arcs*, respectively.

Graphical convention. Nodes are drawn as ovals labelled with the appropriate element (set of formulae of L_B) from H. Arcs are drawn as arrows connecting ovals, and labelled appropriately.

Definition 15. Let $G = (H,P)$ be a graph over L. For $i = 1,2, \cdots$ *let* $p_i = (h_i,l_i,h_i') \in P$.

(1) $w = p_1,p_2, \cdots$ is called a *path* in G iff for $i = 1,2, \cdots$, $h_i' = h_{i+1}$ Then we also write $w = h_1 l_1 h_2 l_2 \cdots$.

(2) w is *finite* iff for some $n \in \mathbf{N}$, p_{n+1} is not constructed. In this case, n is the *length* of w. The empty path is of length 0.

(3) w is a *circle* iff, for some $n \in \mathbf{N}$, w is of length n and $h_n' = h_1$.

Definition 16. Let $G = (H,P)$ be a graph.

(1) G is *acyclic* iff G contains no circles.

(2) $h \in H$ is an *initial node* iff $\{(h_1,l,h) \in P\} = \emptyset$.

(3) G is *finitely based* iff G has only finitely many initial nodes.

(4) G is *finitely branched* iff for each node $h \in H$, $\{(h,l,h_1) \in P\}$ is finite.

Definition 17. Let $G = (H,P)$ be a graph, let $c \subseteq L_B$ be a case, and let $h \in H$ be a node in G.

(1) We define the *subgraph of G rooted at h as* $G' = (H',P')$, where $H' \subseteq H$ is given by

$$H' = \{h' \in H \mid \exists \text{ a path in } G\ h_i l_i h_j l_j \cdots h' \ \&\ h_i = h\},$$

and $P' \subseteq P$ is given by

$$P' = \{(h,l,h') \in P \mid h,h' \in H'\}.$$

(2) We say that $h \in H$ *satisfies* c *iff* $\exists$ *a finite path* $h_1 l_1 h_2 l_2 \cdots h'$, such that $c \subseteq h' \ \&\ h = h_1$.

(3) We say that a graph G *satisfies* c *iff* $\forall$ initial nodes h, h satisfies c.

(4) $h \in H$ is called a *choice point* iff $| \{(h_1,l,h_2) \in P \mid h_1 = h\} | > 1$.

(5) We denote the set of *choices available at* h by $\tilde{h} = \{h_j \mid (h_i,l,h_j) \in P \ \&\ h_i = h\}$.

Using definitions 15 through 17, we can now define the projection of a plan from a case respecting a set of domain constraints. To repeat, the idea is that the graph structure defined contains a given initial case as its initial node, and each node in the graph contains a case reachable under e-element occurrence. Under the interpretation of a case as a belief state for the agent, the initial case describes an initial state of belief, and cases in the graph reachable from the initial case describe future possible states of belief for the agent. The arcs in the graph denote transitions from one state of belief to another, and these transitions can be realised through the actual execution of the actions that correspond to the e-elements labelling the connecting arc.

The term *projection* is used as in Charniak and McDermott (1985) to refer to the "inference" from what's the case at one time to what's the case at another.

Definition 18. Let $P = (B,E;F,A)$ be a plan, let $c \subseteq L_B$ be a case, and let $X \subseteq \Pi(L_B)$ be a set of domain constraints. We define the *assumptive projection of P from c (respecting X)* as the graph $S_{PX} = (H,R)$, where the set of nodes, H, is given by

$$H = \{c' \subseteq L_B \mid c\ r_P^*\ c'\},$$

and $r_P \subseteq \Pi(L_B)\times\Pi(L_B)$ is given by

$$c_1 \, r_P \, c_2 \iff \exists\, G \subseteq E\colon c_1 \overset{X}{\underset{G}{\Rightarrow}} c_2,$$

and the set of arcs, R, is given by

$$R = \{(c_1,G,c_2) \in (H\times\Pi(E)\times H) \mid \exists\, G \subseteq E\colon c_1 \overset{X}{\underset{G}{\Rightarrow}} c_2\}.$$

Definition 18 uses the notion of a step to construct a reachability relation r_P on cases. Under transitive closure, this relation can be used to find all the cases reachable from an initial case, c. This is how the set of nodes, H, is derived. The set of arcs, R, is built up by finding nodes in H that are reachable from each other in a single step.

An exactly analogous definition for the *realistic projection of P from c (respecting X)* can be given, simply by replacing the assumptive step specification $\overset{X}{\underset{G}{\Rightarrow}}$, with the realistic step $\overset{X}{\underset{G}{\rightarrow}}$.

We now suggest a definition of *sound* execution advice. The idea is that a plan's advice must contain the information required to remove harmful residual non-determinism within the plan. The advice should not restrict legitimately causally independent actions from occurring concurrently, but it should prevent planned actions from occurring in an order permitted by the causal structure of the plan but unintended by the agent. A classic example of this phenomenon occurs in blocks-world tower construction problems. For example, given the problem of creating a tower with block C on the bottom, block B in the middle, and block A on top, the plan construction reasoning must order the two required stack actions to reflect its overall goals. To see this, assume that all blocks are initially clear and on the table. If a plan calls for stacking A on B, and B on C, then *either* stack action can actually proceed from the initial state. It is not an ordering enforced by *teleology* that requires the stacking of B on C before A on B. Rather, it is the agent's intention regarding overall plan execution outcome that directs the sequencing of the two actions. In Sacerdoti's (1975) Noah system, the reasoning which introduced this ordering resided in the *resolve conflicts critic*. The definition given below captures the conditions under which a strict before order *must* be introduced between mutually enabled e-elements. This

definition formalises the operation performed by Noah's critic.

It is possible to think of the execution advice as providing a heuristic about which way to proceed with the execution of the plan. It is also possible to think of the execution advice as scheduling information which directs the plan execution system in its selection of which action to attempt, should a number of actions have concession simultaneously. However, the execution advice is not limited to agent-executable actions. The projection of a plan makes no reference to event executancy, and any two sets of e-elements can be included in the before order.

Definition 19. Let $P = (B,E;F,A)$ be a plan, let $c,c' \subseteq L_B$ be cases, let $X \subseteq \Pi(L_B)$ be a set of domain constraints, and let $S_{PX} = (H,R)$ be the assumptive projection of P from c (respecting X).

(1) We say that the execution advice A is *sound with respect to* c' iff $\forall h \in H$, if h is a choice point and $\exists h' \in \bar{h}$, such that h' satisfies c', then either

a) $\forall h' \in \bar{h}$, h' satisfies c' or

b) $\forall h' \in \bar{h}$, if h' does not satisfy c' then $\exists$ $(G',G) \in A$ such that

i) $(h,G,h') \in R$ and

ii) $(h,G',h'') \in R$ & h'' *satisfies* c'

Essentially, this definition says that for all choice points in a plan's assumptive projection, if there is *any* hope for success at the choice point, then either all choices lead to success, or for each choice point that could lead to failure, there is advice about another possible alternative, such that the suggested alternative *can* lead to success. In essence, when there is still hope for success, the advice prevents the wrong choice from being made.

4.5. Assumptions and reasons.

This section defines various modes of existence for formulae of L_B. The basic idea is that, given a case c, a plan net P, a set of domain constraints X, and an L_B formula p, we can classify p as *in* c, or *out of* c, and in addition say whether or not there is a *reason* for this formula being in or out.

Notice that this is a more symmetrical relationship than commonly occurs in RMS formalisms. Here, we are interested in giving a reason for a formula's *out*-ness, as well as its *in*-ness. In this framework, to give a positive reason for an *in* formula is to justify *belief* in it; we consider all other *in* formulae to be *assumptions*. To give a reason for p being out is to give reason to *disbelieve* it; if no such reason can be given, the formula is simply *not believed*. The difference between lack of belief and disbelief is missing from standard RMS accounts of belief and reason. The full relationship between RMS ideas and the definitions we give is discussed in chapter 6, section 4.

Definition 20. Let $c \subseteq L_B$ be a case, let $P = (B,E;F,A)$ be a plan, let $X \subseteq \Pi(L_B)$ be a set of domain constraints, and let $p \in L_B$ be a formula of L_B.

(1) *p is in c* iff $p \in c$.

(2) *p is out of c* (or *p is not in c*) iff $\{p\} \cap c = \emptyset$.

(3) Let p be in c. If $\exists\, e \in E_P\ \exists\, c' \subseteq L_B$: $c' \xrightarrow[e]{X} c$ & $\{p\} \cap c' = \emptyset$ then we say that *p is believed in c with reason e* (or *p is a belief with respect to c and P*), otherwise we say that *p is assumed in c* (or *p is an assumption with respect to c and P*).

(4) Let p be out of c. If $\exists\, r \in R$: $\mu_X(c,\{p\}) = R$ & $r \neq \emptyset$ then we say that *p is disbelieved in c with reason r* (or *p is a disbelief with respect to c and X*), otherwise we say that *p is not believed in c* (or *p is a non-belief with respect to c and X*).

Often, if the plan P, case c, and domain constraints X are clear from the context of the discussion, we will omit reference to them.

According to these definitions, a given formula is *in* a set of formulae if it is a member of the set; it is *not in* if it is not a member of the set. Definitions 20.3 and 20.4 capture the idea of providing *reasons* for a formula being *in* or *out*. A reason for being *in* is formalised as an e-element which under realistic application can produce the desired formula. Informally, we

sometimes say that the formula can be *discharged* by an e-element that is a reason for it. A reason for being *not in* is a reconciliation set which would have to be removed from the set of formulae if the individual formula were to be added. This is formalised using μ: if the case which the formula is not in would become inconsistent by adding the formula, then there is a reason not to add it. The reason is one of the reconciliation sets that it would be necessary to remove, were the single formula to be added. The empty set, ∅, is not considered to be an adequate reason for disbelief.

These definitions classify every formula of L_B according to the formula's "belief" status. If the formula is *in*, with a reason, then it is *believed* with that reason. If the formula is *in*, but does not have a reason, then it is *assumed*. If it is not in, and has a reason to be not in, then it is *disbelieved* for that reason. The default catch-all case is when the formula is not in, and has no reason to be not in: in this situation, we say that the formula is simply *not believed*. We can expect that the huge majority of L_B formulae will be *not believed*; only a few will actually be *disbelieved*. Notice that all these definitions apply not only to a given b-element, p, but also to a given case, c. It will often happen that p is believed in c with reason r, disbelieved in c' with reason x, and assumed in c''. There is no inconsistency here, since c, c', and c'' are different cases, and there are no constraints on what each may contain.

We will continue to speak loosely of cases $c \subseteq L_B$ as being states of belief, since the analysis presented above applies only to individual L_B formulae. When more resolution on the status of any individual formula within a case is required however, we will be precise about whether the formula is *believed* or *assumed*.

4.6. Defining problems.

With the definitions of an operator, plan, belief consistency and the two forms of projection in hand, we can now formalise *problems*. The definition is straightforward: a planning problem consists of a set of operators, a set domain constraints, and two cases -- the problem's initial case and goal case.

Definition 21. A 4-tuple $R = (O,X,i,g)$ is called a *planning problem* iff

(1) O is a set of operators;

(2) $X \subseteq \Pi(L_B)$ is a set of domain constraints;

(3) $i \subseteq L_B$ is called the *initial case;*

(4) $g \subseteq L_B$ is called the *goal case.*

Now for a plan to be a potential solution to a planning problem, it must satisfy 3 conditions. First, its assumptive projection must satisfy the goal case, according to definition 17.3. Second, the plan's execution advice must be sound with respect to the goal case of the problem, according to definition 19.1. Third, the plan must be composed from the operators given in the problem specification. That is, we allow no imaginative solutions, in which a problem solver attempts actions that are not permitted by the problem specification. This last condition is formulated simply by requiring that all e-elements in the plan can be derived from the e-elements in the provided operators. This derivation is possible only through substitution of constants for the operators' e-element variables.

Definition 22. Let $P = (B,E;F,A)$ be a plan, let $R = (O,X,i,g)$ be a planning problem, let $S_{PX} = (H,R)$ be the assumptive projection of P from i, respecting X, and let $E_O = \bigcup_{(B',E';F',A')\in O} E'$ be the set of all e-elements in the operators in O. P is called a *potential solution* to R iff

(1) S_{PX} satisfies g.

(2) The execution advice A is sound with respect to g.

(3) $\forall e \in E_P \ \exists e' \in E_O \ \exists$ *a valuation* β: $e = \hat{\beta}(e')$. We say that the plan P is *composed* from the e-elements contained in the operators in O.

Notice that definition 22.1 says nothing about plan minimality or optimality. Such guarantees could only be made by comparing path length in the projection S_{PX} with path lengths in

the projections of other plans. Notice also that definition 22.3 doesn't say that operators must be used *in toto;* that is, a legal plan can result from e-elements being assembled from various operators. The construction techniques we present (later, in chapter 6) do not look into an operator, in an attempt to extract useful component events. Our plan construction techniques use operators as atomic, indivisible units. Thus, the definition of potential solutions as given in definition 22 might admit plans that these techniques are unable to generate.

4.7. Execution.

We must now explain the execution of a plan. So far, we have defined operators, plans, domain constraints, belief consistency maintenance, plan projection, assumptions, planning problems, and specified when a given plan is a potential solution to a problem. But a plan is only a *potential* solution, and not an *actual* one: until the plan is executed, the desired effects will not be realised. The flavour of exposition most suited to explaining the execution of a plan is an *operational* one, as employed in the area of operational semantics for programming languages. It is possible to provide an operational semantics for a programming language by specifying an interpreter for the language (for a readable introduction see Wegner, 1970). This sort of operational approach is also used by Georgeff, et al. (1985), in explaining the execution behaviour of their plan structures.

A discussed in chapter 2, one of the most basic ideas in planning work is that an agent has access to a *world model.* Recall that this world model is a database containing assertions which describe features of the agent's environment. In the terminology developed in this chapter, such a world model contains a *case,* composed of formulae of L_B. It is important to keep in mind the distinction between the current case, as exists in the agent's world model, and possible cases that occur in a plan net's projection. The world model is special, in that it is "directly connected" to the environment through the agent's sensory modalities. This is not to say that the world model will never be incorrect. Indeed, we expect that quite often the agent's beliefs will not accurately reflect the state of its environment. But it is important to remember that the world model determines the agent's *current* state of belief regarding its environment, and each case in a projection is a *possible* state of belief for the agent at some point in the future.

We define the execution of a plan $P = (B,E;F,A)$ using a function @. When this function is given a formula of L_E, it returns the address of the routine corresponding to the formula, or returns *undef*, indicating that the corresponding routine does not exist. Mathematically, we might view @ as a partial function from L_E to "machine addresses". It would perhaps be an interesting exercise to attempt to provide a more "denotational" sort of semantics for plan execution, in which these addresses form a domain. We use the operational approach here, since it is simpler and thus more easily understood, and also because it seems more in-keeping with general Net Theory philosophy.

This @ function is the way we realise the idea of *event executancy*, as discussed in chapter 2. If @ is defined, then the agent has executancy over the action, since a routine does exist by which the interpreter can carry out the action. If @ is undefined, then the environment must have event executancy, since the agent does not. Notice that this has quite far reaching consequences for the plan execution algorithm. In particular, if @ is undefined for a given e-element, and the e-element continues to have concession, the interpreter we present might wait *forever* for the corresponding action to occur. To get around this, we would have to build in some sort of idea about the *duration* of an action, and probably also have to add some measure of action *utility*. We do neither of these here. Thus if an e-element has concession, and the agent does not have executancy (i.e. if @ is undefined), then the interpreter we give will wait until the corresponding action actually occurs.

To illustrate this, imagine that there is an e-element which corresponds to the event of a light turning green at a traffic intersection. If the light is broken, but none of the agent's beliefs are updated to reflect this, the interpreter will wait at the red light forever, awaiting the change to green. It seems that a different sort of reasoning is required for the agent to realise that it has been waiting for something to happen for *some time*. This idea of *some time* is vague, and difficult to capture in *any* formalisation. Note that this is not a problem with the plan net *representation*, but a feature of how the interpreter we present is defined. The interpreter *could* be defined so that it returns *failure* immediately upon discovering that there are no @-defined model-enabled e-elements. Exactly how the interpreter we present works is discussed below.

```
execute( )
  if ∃ valuation β: β̂(goals) ⊆ model then return( success );
  else begin
    /* Step 0: bind the plan as possible in the world model */
    bind-plan( );

    /* Step 1: find model-enabled e-elements */
    ME := {e ∈ E | e is model-enabled};

    /* Step 2: filter according to execution advice */
    ME := filter-using-advice( ME,A );      /* See figure 4.9 */

    /* Step 3: plan failure test */
    if ME = ∅ then return( failure );

    /* Step 4: collect @-defined e-elements */
    AT := {e ∈ ME | @( e ) ≠ undef};

    /* Step 5: execute actions and collect results */
    RE := ∅;
    for each e ∈ AT
       RE := RE ∪ call( @( e ) );

    /* Step 6: update model for actions' internal results */
    model := model ∪ RE - choose( μ( X,model,RE ) );

    /* Step 7: assume the external effects as specified by e⁻ */
    if assumptive then
       model := (model ∪ AT⁻) - choose( μ( X,model,AT⁻ ) );

    /* Step 8: execute plan remainder */
    return( execute( ) );
  end.
```

Figure 4.8: One possible plan net interpreter.

It is important to understand that the idea of event executancy only arises in terms of plan net *execution*, while the effects of an action on the agent's beliefs are captured in plan net *structure* and *projection*. The effects of an action on the agent's beliefs have been formalised in terms of an e-element's external-results postset and internal-results postset. As shown in previous sections, an analysis of these sets allows one to distinguish between the internal and external

results of any given action. With the addition of the @ function, we now have all the components suggested by the epistemology of chapter 2.

It seems useful at this point to try to paint a small part of a larger picture. This picture must include the overall interaction of the agent and its environment. To do so will make subsequent discussion easier.

We suppose that as the agent reasons about how to solve its problems, it also exists continuously in its environment. This means that the agent must maintain the consistency of its world model, as it reasons. The agent does not solve problems in idyllic isolation: the environment changes constantly, and the agent's beliefs must track these changes. So while the agent might be either *forming* or *executing* a plan, it is *always* existing in its environment. During problem solving, new information will often become available from the environment which must figure in the problem's solution. This new information must therefore be integrated with the agent's existing set of beliefs. Recent comments by Sloman (1985, p. 214) support part of the architecture we suggest: "... the perceptual process of absorbing new information and assessing its significance should go on in parallel with plan execution. ... logically there are distinct parallel processes: acting and perceiving."

```
filter-using-advice( Events, Advice )
 begin
  R := Events;
  for each (G,G') ∈ Advice
    if G ⊆ Events & G' ⊆ Events
    then R := R – G';
  return( R )
 end.
```

Figure 4.9: The advice filter.

It seems then that there are three main functions that require explanation: plan formation, plan execution, and belief system monitoring. We delay discussion of plan formation until chapter 6, since it is an extremely complex topic in its own right. Even in chapter 6, we do not explain plan formation in great depth, since the major contributions of this work are in the area of plan representation. However, since we are offering a new representation, it *is* necessary to explain how any given plan in this representation might execute. For completeness, we also provide an account of how the agent can maintain consistency in its world model. These topics are discussed in turn below.

In figure 4.8, we define a *plan net interpreter*, which has constant access to four things:

(1) a current world model (*model*);

(2) a set of domain constraints (X);

(3) a set describing the agent's overall goals (*goals*); and,

(4) a plan to execute ($(B,E;F,A)$).

The world model is used to find e-elements in the plan being executed that have concession. Recall the notation introduced in definition 12.1: Given a case, $c \subseteq L_B$, a plan $P = (B,E;F,A)$, and an e-element $e \in E_P$ which has concession in c, we can also say that e is *c-enabled*. The algorithm makes use of this notation, by treating the world model as a case, and thus referring to those e-elements in the plan which are *model-enabled*.

In figure 4.10, we present a simple belief system consistency monitor, which accesses the latest observations returned by the sensory subsystems. We assume that all required input, be it visual, auditory, tactile, or whatever, is pre-processed by the appropriate sensor interpretation routines. These routines must convert the signals they receive from sensors into a form suitable for direct integration with the agent's existing world model. Thus, we are not attempting to address the "pixels to predicates" problem. It is assumed that a mechanism for translating from sensor information to the formulae of L_B already exists.

The consistency monitor runs continuously. As shown in figure 4.10, it has access to the same world model (*model*) and domain constraints (*X*) as the interpreter. The monitor accesses the latest environmental observations, and integrates these observations with the existing world model formulae. To do this, the monitor uses an algorithm implementing the belief consistency maintenance function. To choose one of the possible reconciliation sets, the monitor uses a function called *choose*, which should embody the same heuristic used by the plan projection algorithm for selecting amongst the possible future belief states that occur in the plan's projection, formalised by ψ. If *choose* selects reconciliation sets by some other means, then plan execution will likely halt. The world model will probably be such that no plan e-elements are model-enabled. Of course, the plan formation reasoning could include e-elements in the plan which take account of all possible reconciliation set selectors, but this seems a bit extreme. As long as the plan formation reasoning uses the same reconciliation set selector as the *choose* function in the belief system monitor, then there will be no problem.

The interpreter of figure 4.8 would operate roughly as follows. Initially, it must check to see if there is any way that its goal set can be satisfied by the current world model. This is true if the goals can be bound so as to become a subset of the model. If an appropriate binding exists, the interpreter exits with *success*.

```
monitor()
  begin
    O := get-sensor-information();
    model := (model ∪ O) - choose( μ( X,model,O ) );
    monitor()
  end.
```

Figure 4.10: The belief system consistency monitor.

If no appropriate binding for the goals exists, then the interpreter begins in step zero by finding a binding for the plan in the world model. The routine *bind-plan* requires no argument, since it operates only on the globally available plan. It binds the plan in terms of constants available in the world model. Not all possible plan bindings need be made -- here, we are satisfied with a single binding. Also, not all variables in a plan need be bound. Only those that can be need be. A binding can be performed by checking all b-elements in the plan which contain variables, and finding appropriate L_B formulae containing constants in the world model. These constants can be substituted for the plan's b-element formula variables. This is how the system can plan to act on the basis of information unavailable at plan generation time. From the plan generation perspective, it suffices to say "what ever the exact value is, here is the general form of what to do", and the exact values get filled in during plan execution.

Step one uses the bound plan derived in step zero to find all model-enabled e-elements. These e-elements are collected in a set *ME* (for Model Enabled), and denote actions that the agent believes to be enabled. This set is filtered in step two. The routine *filter-using-advice* (given in figure 4.9), simply removes those currently enabled e-elements which are precluded from execution by the plan's execution advice. If no model-enabled e-elements exist after all this filtering, the interpreter terminates in step three with failure.

Providing that there are some e-elements to execute, the interpreter collects together those e-elements for which the @ function is defined in a set *AT* (to suggest @). This is done in step four. By finding @-defined e-elements, the interpreter isolates all those e-elements for which the agent has executancy. This set of actions is *executed* in step five. The routine address corresponding to each e-element is found using @, the routine is called, and the results of the routine are saved in a set *RE* (for REsults). *Whatever* the internal results are of the called actions, they must be passed back through *call* so that the world model may be updated appropriately in step six.

When an e-element is evaluated by @, it *must* have all variables occurring in it bound to appropriate constants. We can view the mapping from an e-element to the actual action it denotes

as a process of applying a specific function to given arguments. Much the same view is taken by Moore (1985, p. 341), and by Konolige (1980, p. 34, following Moore, 1980). In our case, the *function* is the predicate of L_E in the e-element, and the arguments are the constants of L. Requiring arguments to be constants forces the e-element action description to be *executable*. This requirement could be syntactically enforced in the plan net formalism by ensuring that any variables occurring in an e-element e, also occur in $^{\pm}e$. However, such a requirement is unnecessarily restrictive in cases where @ is undefined. It seems reasonable to say that if any variables remain in an e-element when evaluated by @, then the returned value is *undef*, indicating that the e-element does not constitute an executable description of an action. A similar view on what constitutes an executable action description is given by Nilsson (1985).

Note that the execution in step five may or may not be in parallel, depending on how the interpreter is actually implemented. As defined, it is only *possible* to execute the e-elements in parallel. If such parallel execution were *necessary* for the successful execution of a plan, the interpreter might fail. The plan net formalism as yet has no means for specifying *required* concurrency. We return to this issue in chapter 9.

In step six, the internal results of all executed actions are added to the world model via the belief consistency maintenance function. μ is passed the domain constraints, current world model, and the actual internal results produced by the executed actions. These results become new beliefs.

Step seven checks a flag, *assumptive*, to see if any assumptions regarding the physical outcomes of the executed actions should be made. If this flag is set (as seems reasonable most of the time), then the external-results postsets of the actions are added to the world model, and μ is used to enforce consistency. Thus the plan construction algorithm could employ an *assumptive projection* when considering plan behaviour, and set the *assumptive* flag to indicate this to the interpreter.

Step eight simply continues plan execution, and returns the final result of executing the plan.

```
μ( X,C,N )
 begin
  /* X: domain constraints; C: current case; N: new observations */
  /* Step 0: initialise */
  I := R := ∅;

  /* Step 1: determine possible inconsistencies */
  for each n ∈ N
   for each x ∈ X
    I := I ∪ unify-element-all-possible-ways(x,n);

  /* Step 2: delete new observations */
  for each n ∈ N
   for each i ∈ I
    if n ∈ i then I := (I - i) ∪ (i - n);

  /* Step 3: find existing beliefs not in observations */
  O := C - N;

  /* Step 4: find all bindings for constraints */
  for each i ∈ I
   R := R ∪ unify-set-all-possible-ways(O,i);

  /* Step 5: factor constraints into deletion sets */
  P := factor(R);                % See figure 4.12

  return(P);
 end.
```

Figure 4.11: Algorithm for the belief consistency maintenance function.

In figure 4.11, we give an algorithm for the consistency maintenance function, μ. An example of the behaviour required of this function has already been given in section 4.3 above. The appendices also contain the code for a Prolog implementation, with sample runs in a variety of domains. Here, we only briefly describe the algorithm presented in figure 4.11. Refer to the Prolog implementation for a more computationally effective formulation.

In step one a set, called *I*, is formed which contains domain constraints. Each domain constraint in *I* is formed from one of the domain constraints in *X*, bound using one of the new observations in *N*. This binding is accomplished through the use of a function *unify-element-all-*

possible-ways. This function accepts two arguments: the first argument is a domain constraint x, and the second is a new observation n. The function returns a set of domain constraints, obtained by unifying the observation n in all possible ways with the domain constraint x . In the implementation presented in the appendices, this is achieved using a combination of Prolog's pattern matching and backtracking mechanisms.

In step two, these derived constraints are simplified by removing all the new observations. This ensures that subsequent attempts to satisfy the constraints will never consider removing a new observation.

Step three removes all the new observations from the "current" case, C. This guarantees that constraints will not be satisfied by elements of C which also happen to be in N. After this, we have O containing those L_B formulae that are unique to C, and I containing partially bound domain constraints -- bound in every possible way in terms of each new observation. Thus, I represents those constraints which might be violated in C, and O represents those formulae unique to the current set C. Notice that this algorithm assumes that both C and N are initially consistent. If either set is initially inconsistent, the algorithm is not guaranteed to succeed.

Step four forms a set R from the L_B formulae in O and each domain constraint in I. Each element of R is a domain constraint obtained from a constraint i in I. A constraint in R is obtained by binding the remaining variables of i in all possible ways, using constants found in the case O. This is performed by the function *unify-set-all-possible-ways*, similar to the function used in step one. Both functions require some sort of underlying unification machinery. We do not present algorithms for performing this unification, since the literature abounds with variations on the unification theme. Suffice it to say that some sort of unification must be performed.

Step five factors the remaining (fully bound) constraints into reconciliation sets. Since this step is rather interesting, we present an algorithm for it in figure 4.12. Given a set of domain constraints, this function returns a set which contains sets of possible deletions. Each possible deletion is formed by selecting one element from each of the domain constraints given in the function's argument. Thus P is a set of formulae from C, such that if any one of the sets is

removed from C, the union of C and N will be consistent with respect to X.

4.8. Summary.

This chapter has introduced the formal notions and notation required to discuss plan nets, problems, and the conditions under which a plan net offers a potential solution to a problem. In order to do this, we have defined basic nets, operators, plans, domain constraints, the consistency maintenance function, assumptions, and plan net projection. All these definitions have been motivated with simple expository blocks-world examples.

A plan net interpreter and belief system monitor have been presented. The interpreter accepts a plan, and carries out the actions the plan specifies. During execution, the belief system monitor relies on an algorithm implementing the the belief consistency maintenance function to update the world model in the light of newly sensed information. A basic algorithm has been presented by which the agent may perform this belief consistency maintenance. For those who want to see an actual program, the appendices contain the Prolog source code of an

---

```
factor( X )
  if X = ∅ then return({∅});
  else begin
    A := ∅;
    M := car(X);                    % We assume lisp-like car and cdr functions
    R := factor( cdr(X) );          % to access first and rest of X respectively
    for each m ∈ M
      for each r ∈ R
        begin
          N := r ∪ {m};
          A := A ∪ {N};
        end;
    return(A);
  end.
```

Figure 4.12: Algorithm for factoring constraints into reconciliation sets.

---

implementation and some sample runs.

In the next chapter, examples are given which better demonstrate the power of the plan net formalism. While most of these examples are based on much richer domains than the simple blocks-world, some blocks-world examples *are* presented to facilitate comparison with previous plan representations.

Chapter 5: Example Plan Nets.

I do not greatly care whether I have been right or wrong on any point, but I care a good deal about knowing which of the two I have been.

– Samuel Butler (II)

5.1. Chapter overview.

In this chapter we present a number of example plan nets. Each example generally consists of the basic plan net, a set of domain constraints, an initial case, and an assumptive plan net projection. We present each example in two parts: first, a small bit of narrative describing the problem to which the plan net is a solution; and second, a graphical version of the net and its projection, each drawn according to the conventions set out in chapter 4. Each example is given its own section, with matching figure number. To arrange this, there will be no figure 5.1. The first plan net and projection are graphically presented in figure 5.2.

In addition, we adopt the convention that variables begin with upper case letters, and constants begin with lower case letters. This permits more intuitive variable and constant names to be used than if we were to stick exactly with the notation of the last chapter. Of course, nothing fancy is happening here: each variable and constant we introduce can be considered to stand for some v_i or a_j in L.

We will always use S_0 as the initial case from which to project a plan net. Where it is relevant, we will indicate that the agent has executancy over an event by placing an at-sign "@" in the box representing the e-element that denotes the event.

As a point of interest, notice that each formula in each projection's initial case is an *assumption*, according to definition 20.3. Each projection begins with a set of assumptions (possibly ∅); the given plan's possible behaviours are determined from this initial set.

Also note that in the accompanying text, formal symbols will be used as the names of the objects they denote. Otherwise the reading process (writing too) is extremely tedious. For instance, instead of saying "the object denoted by the symbol *larry*", we will instead say simply "*larry*".

All plan net execution advice can be assumed to be ∅ unless given explicitly, in which case it will be listed as a set of ordered pairs with the net. All example nets are intended to be plans for our one distinguished agent.

5.2. Crossing the road.

This plan net describes the following behaviour. Our agent is initially at some location, *l1*. The goal is for the agent to be at some other location, *l2*. There are two preconditions for the agent moving from *l1* to *l2*. First is that the agent is at *l1*, and second is that the traffic control light is *green*. The scenario might be one in which the agent is on one side of a street, and wishes to cross to the other side. Being law-abiding, the agent will cross only when the pedestrian control light is green.

The example calls for the possible occurrence of only three events: the agent moving from *l1* to *l2*, the signal changing from *red* to *green*, and the signal changing from *green* to *red*. According to the structure indicated in the figure, in order for the light to change from *green* to *red*, it must first be *green*. After changing, the light will be *red*. Similarly for the colour change in the opposite direction.

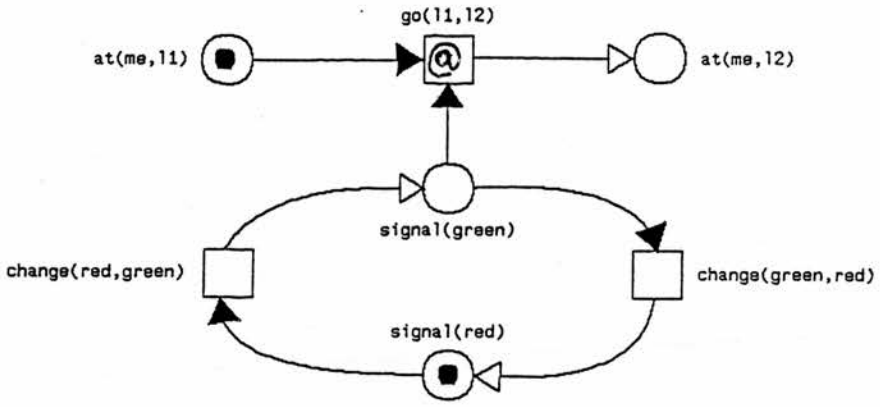
The only event for which the agent has executancy is moving from one side of the road to the other. The agent does not have executancy for either of the light changes. Notice that this does not affect the plan's projection. It is only at plan *execution* time that executancy is of concern.

The initial case given has the agent assuming itself to be at location *l1*, and assuming that the signal light is *red*. The domain constraints for this example are

{ {at(O,L1), at(O,L2) }, {signal(C1), signal(C2)} }

and indicate to the agent that any object is only ever *at* one location, and that the traffic signal light is only ever one colour. An assumptive projection is given which uses this initial case and these domain constraints. The projection has the initial case labelled as S_0. Three other cases are reachable from S_0. The projection indicates that the light can change state an infinite number of times before any movements by the agent occur. The agent can only move when the signal light is green; that is, the movement action is only possible from S_1. Once the agent does move, an assumption about its new location forces a change in belief state, and now an infinite number of signal light state changes can occur, this time, with the agent safely at location *l2*.

According to the definitions of the last chapter, all the formulas in all the cases in the projection are assumptions, since none can be discharged by unapplying any of the e-elements contained in the given plan net.



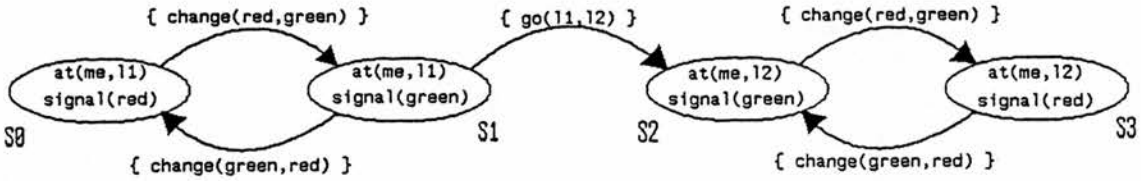


Figure 5.2: Crossing the road.

5.3. Hammering a nail.

In this example the agent is responsible for hammering a nail until the nail is no longer protruding from the surface into which it is being hit. The plan is rather single minded, and will allow the agent to beat away *forever*, if the nail only moves into the material an infinitely small amount with each impact, and never bends.

There are three events of concern itemised in the plan net. The agent can *hit* the nail and *lift* the hammer, preparatory to another hit. The conditions linking the two hammer movements are the hammer being up, and the hammer being down. Lifting causes the hammer to be up, and hitting causes the hammer to be down. According to the *hit* e-element, hitting also causes the nail to be in. Thus this *hit* e-element models the perfect action of striking a nail once and having it go straight in. The third event modelled is that of the agent sensing that the nail is sticking *out*. As shown in the figure, the agent does have executancy for the actions of hitting and lifting, but it does not have executancy for the action of seeing the nail being *out*. This last e-element has an internal-results postset of {*nail(out)*}, and thus has the internal result of the nail being believed to be up. This means that following the occurrence of the denoted event, the agent *will hold a belief* about the nail being up. This is in direct contrast to the other two actions, since according to the plan net, following their occurrence, no beliefs about the hammer being up or down will be gained. When building the assumptive projection, we simply *assume* that the actions will occur as described, and that while executing the plan, enough information will be received from the environment to cause the appropriate beliefs to be held. The actions by which the agent becomes aware of the hammer being up or down are not explicitly modelled in the plan net given.

The domain constraints

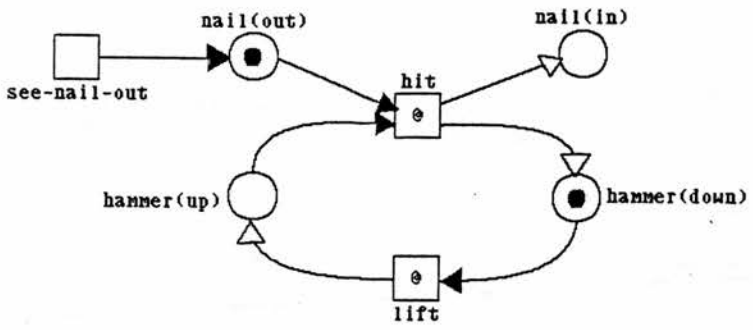
{ {nail(X1), nail(Y1)}, {hammer(X2), hammer(Y2)} }

specify that the nail has only one status (in the example, either *out* or *in*), and similarly, that the hammer is in one of two positions (here, either *up* or *down*). The initial case has the hammer being *down*, and the nail being *out*. In the plan net's assumptive projection, four states are possible, but only two are of direct concern: S_0, and S_3. In S_0, the hammer is assumed to be *up*,

and the nail is believed to be *out*. In S_3, the hammer is assumed to be *down*, and the nail is assumed to be *in*. As is clear from the projection, there are two main steps between S_0 and S_3. Since the actions of seeing and lifting are causally independent, they can occur simultaneously, and be applied as a set to S_3, generating S_0. Notice that two other routes from S_3 to S_1 exist: these correspond to possible interleavings of the *see* and *lift* actions. Since these actions are causally independent, they can execute in any order. On the return path, from S_0 to S_3, only the action of hitting is possible.

The assumptive projection indicates that hammering could go on forever, as seems reasonable, given the simple plan. Following any given hit, it is conceivable that another action of seeing the nail *out* could occur, and it would then be necessary for the agent to make another attempt to sink it *in*.

Only one reasoned belief exists in all projection cases: *nail(out)*. This is because its presence can be discharged by the *see-nail-out* e-element. It is not an assumption simply because it has a legitimate (sensory) justification. All other b-elements in each case *are* assumptions, either because they are given initially, or because they have been introduced by the assumptive projection mechanism on the basis of the denoted actions' declared physical results.



Domain Constraints = { { nail(x), nail(y) },
{ hammer(w), hammer(z) } }

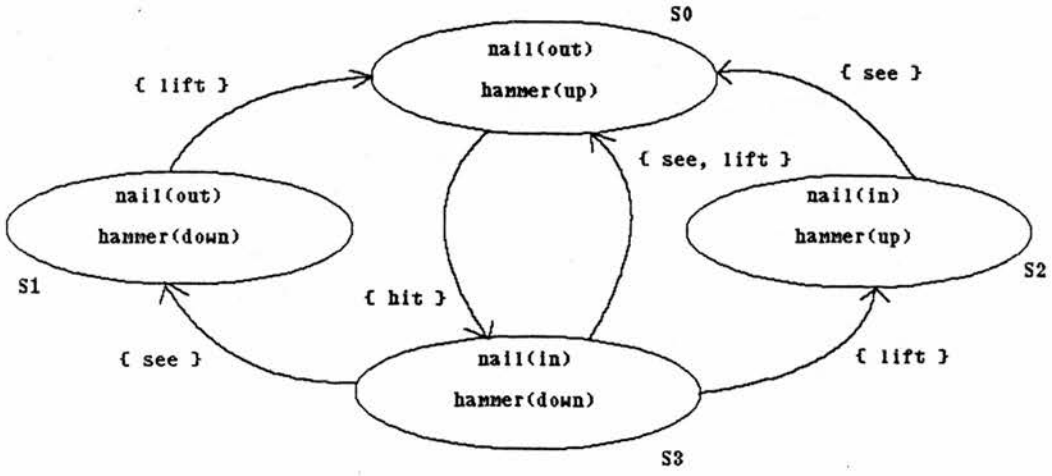


Figure 5.3: Hammering a nail.

5.4. Telling Larry about the meeting.

Picture the following scenario. The agent is at its desk, working on getting its Ph.D. dissertation finished. It has the goal of communicating to some other agent, *larry*, the fact that a discussion about its upcoming submission is to be held at noon. Now while we haven't the mechanisms in place to discuss *larry* as an aware and reasoning individual, and thus can't plan to have *larry* come to *know* or *believe* anything, we can phrase the problem as one of getting *larry* to receive the required information. We suppose that there are two distinct ways of achieving information receipt. The first is to *tell larry* the information; and the second is to leave *larry* a *message*. Leaving an agent a message may not in reality seem like a guaranteed method of communicating with them, but for this simple example, it will suffice.

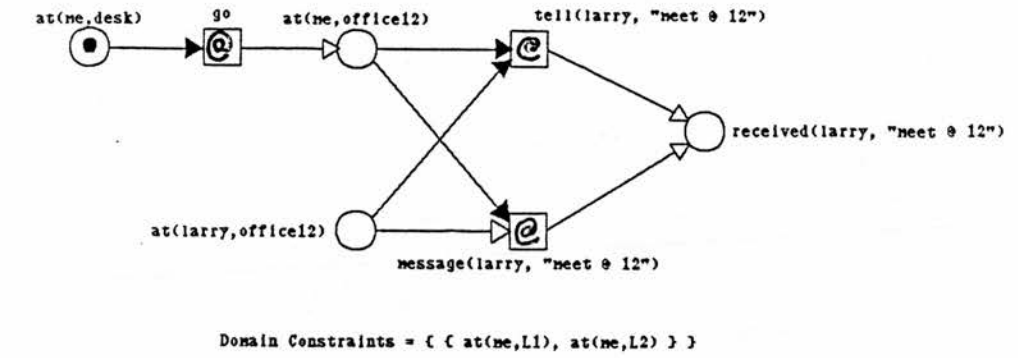
The plan we present for solving this problem calls for the agent to *go* from its *desk* to *office12*. In order to *tell larry* the message, the formula *at(larry,office12)* must be in. To leave the message, this formula must be out. In either case, the formula *at(me,office12)* must be in. The modelled results of both the *tell* and *message* actions is that *larry* has received the required information.

The plan net is designed so as to correctly function from one of two possible cases, as shown. In the first, the agent assumes itself to be at its *desk*, and assumes that *larry* is in *office12*. In the other case, the agent assumes only that it is at its *desk* (this is the case depicted graphically in the plan net of figure 5.4). We provide two assumptive projections of the same plan from each of the two initial cases. A projection of the plan from either case demonstrably results in a case in which *larry* is assumed to have received the message.

The single domain constraint

{ at(me,L1), at(me,L2) }

specifies that the agent can only be at one place at one time. All b-elements occurring in the projected cases are assumptions, since there is no way to discharge any of them using e-elements of the given plan net.



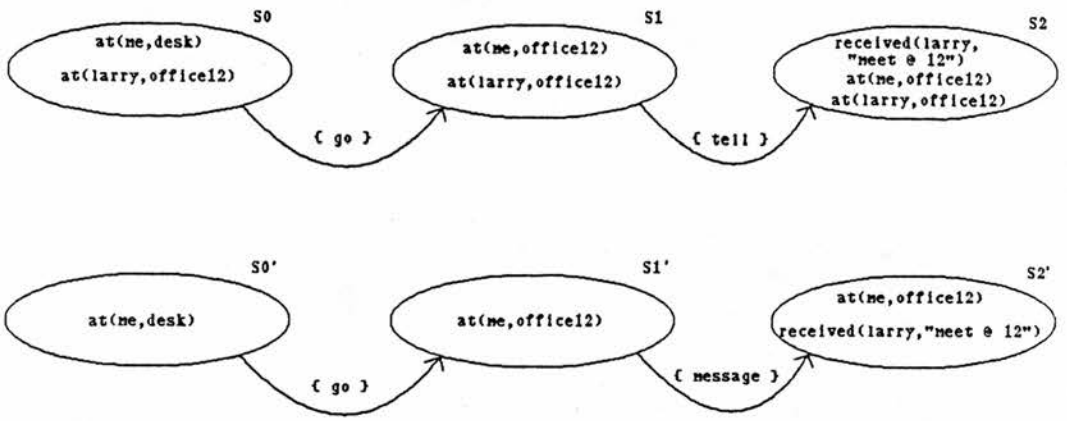


Figure 5.4: Telling Larry about the meeting.

5.5. Phoning Mike.

In this example the problem is to call another agent, Mike, using the telephone. Initially, our agent assumes that it has a certain piece of *paper*, that its eyes are *closed*, that the lights are *on*, and that written on the piece of paper is a *number*.

The plan given in figure 5.5 calls for the agent to *open* its eyes, and *look* at the paper. These actions can occur in any order, or in parallel. Following this, the agent expects to be looking at the paper, and also expects its eyes to be open. The plan does not include a description of the actions by which these beliefs will actually be gained. The projection is assumptive, and thus uses the external-results postsets of e-elements to generate assumptions about the external outcome of actions.

The e-elements in the plan can be understood as follows. The *look* e-element denotes the action that is the movement of the agent's visual apparatus so as to be directed towards the paper. The *open* e-element denotes the action of the agent opening its eyes. Under assumptive projection, these two actions cause a state of belief in which the agent assumes itself to be looking at the paper, it assumes its eyes to be open, the lights to be on, the number to be written on the paper, and it also assumes itself to have the required piece of paper. Three of these assumptions persisted from S_0, and two are new, added by the assumptive projection mechanism. Four of the formula *in* S_1 are just those required to enable the *read* e-element. The single internal-result of the denoted action is a belief of the form *value(num,N)*; that is, following occurrence of the denoted action, the agent will believe something about the value of the number written on the paper.

A formula regarding the value of the number must be *in* to enable the *dial(N)* e-element. This e-element denotes the action of using the value of the number to dial a phone. This is the sort of action that we were discussing in the last chapter in terms of "executability". In order for the e-element *dial(N)* to constitute an executable description of the action of dialing a telephone, the variable N must be bound. In terms of net projection, a binding is irrelevant: the *dial* e-element is enabled just when all of its enable-in preset is *in* the given case, and its enable-out preset cannot possibly be *in* the given case (no binding exists by which the enable-out preset can

be made *in*). These conditions do hold during plan net projection, since the formula *value(num,N) is* in S_2, and thus the *dial(N)* e-element has concession in this case. After its occurrence, the agent will be in the state of belief shown in S_3: it will believe and assume everything it did in S_2, plus it assumes that it will hear the phone ringing. When the plan is executed, if the dialing action does not result in the phone actually ringing, then the action described by the *dial(N)* e-element has not been realised. All this means is that the planned action of dialing the phone and having it ring never actually happened.

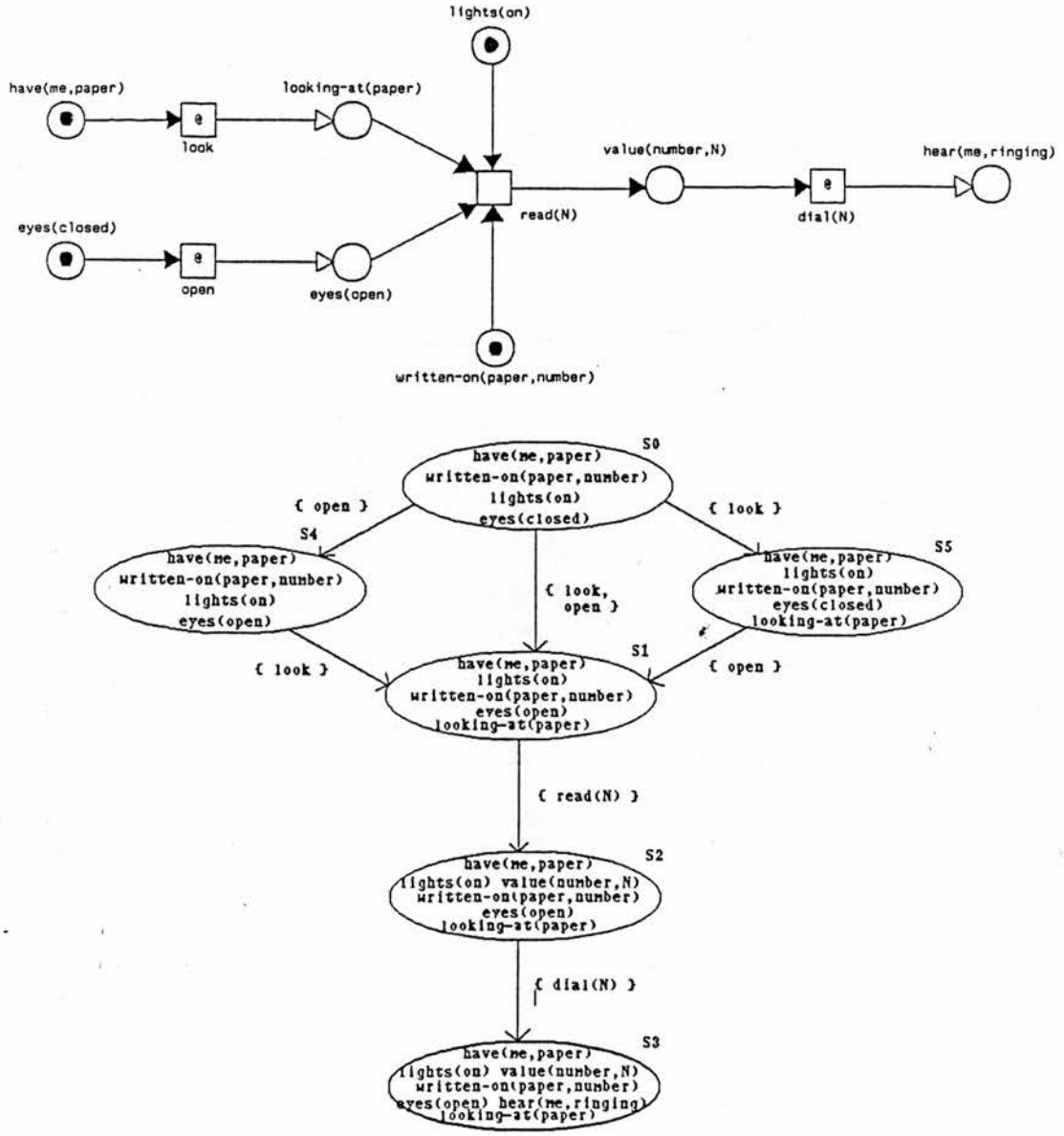


Figure 5.5: Phoning Mike.

5.6. Stacking blocks.

This problem is one of the blocks world classics. Initially, the agent assumes that *a*, *b*, and *c* are blocks, that *b* and *c* are clear, that *a* and *b* are on top of something called *t*, that this thing *t* is a *table*, and that *c* is on top of *a*. The target state of belief is one which contains the e-elements *on(a,b)* and *on(b,c)*. Despite the apparent simplicity of the problem, the plan and projection are actually the most complicated we have yet seen.

The domain constraints required for this problem are those first introduced in chapter 4. We repeat them below.

$$\{\{\ on(X_1,Y_1),\ clear(Y_1),\ block(Y_1)\ \},$$
$$\{\ on(X_2,Y_2),\ on(X_2,Z_2),\ block(X_2)\ \},$$
$$\{\ on(X_3,Y_3),\ on(Z_3,Y_3),\ block(Y_3)\ \},$$
$$\{\ block(X_4),\ table(X_4)\ \},$$
$$\{\ on(x_5,y_5),\ on(y_5,x_5)\ \}\}$$

These constraints are used to indicate that blocks cannot both support an object and be clear; that blocks can be supported by only one other object; that blocks can only support one other object; that nothing is both a block and a table; and that two things cannot be on each other. The second last domain constraint is actually unused in the plan projection, since none of the operators' postsets contain any formulas of the form *table(X)*, and are therefore unable to violate it.

The plan net contains four e-elements: three denote the required block movement actions, and the fourth denotes the action of inferring the clearness of block *a*. Often, when the blocks world is axiomatised, the property of being "clear" is included as a basic, un-inferable property of an object. This seems unreasonable, so we've constructed our plan net solution rather differently. Here, the "clearness" of an object is determined by the agent using the object's *type*, and by whether or not the object supports any others. The *infer-clear* e-element given in figure 5.6 specifies that if no b-element of the form *on(X,a)* is *in*, then block *a* is clear. The

corresponding e-element for tables would say that if an object is a table, then the table is simply (always) clear. This encodes the idea that tables have infinite capacity. In the example, this latter e-element for deriving beliefs about the clearness of tables is not required, since an initial assumption about the table being clear has already been given.

In this example, we have taken a few liberties with the plan net projection. Each assumptive projection case is drawn graphically, according to the following conventions. Blocks are drawn as single (rather malformed) letters, and the table is drawn as a long thin horizontal (unlabelled) line. A clear block is indicated by a single letter with a small horizontal line immediately above it. If no b-element regarding the clearness of a block is in a case, then no line is drawn above the character representing the block. The hope is that these conventions make the projection easier to understand.

This is the first plan net we have seen that has any execution advice. As shown in the figure, the plan's advice consists of two ordered pairs: one calls for the execution of {*move(c,a,t)*} before {*move(b,t,c)*}, and the other calls for {*move(b,t,c)*} before {*move(a,t,b)*}. It is intended that an e-element of the form *move(X,Z,Y)* describes the action of moving block *X* from *Z* to *Y*. Thus the execution advice specifies that block *c* should be moved from *a* to *t* before block *b* is moved from *t* to *c*. Notice that in the initial case of the assumptive projection (S_0), both are possible actions. That is, there are two steps out of S_0: one step of {*move(b,t,c)*} and another of {*move(c,a,t)*}. The execution advice is intended to describe the latter step as less appealing than the former. Recall the definition of *sound* advice: when hope remains for reaching a case which satisfies the given goals, the advice must prevent a step from being taken which leads to a projection sub-graph that cannot satisfy the goals. This is precisely the situation which obtains in figure 5.6. Moving *b* onto *c* from *t* in S_0 locks the net, such that no further progress is possible. In just the same way, the two steps from S_3 of {*move(a,t,b)*} and {*move(b,t,c)*} are ordered in the execution advice so as to indicate that stacking *b* on *c* should come before stacking *a* on *b*. If given only the plan net, an execution system may make the wrong choice of which set of actions to execute when given an alternative. However, when sound execution advice is included, no false moves should be made.

Notice that all the e-elements in the net indicate that the agent does have executancy for the corresponding events.

The projection has been constructed using the reconciliation set selection function suggested in chapter 4. Recall that the idea presented there was that beliefs or assumptions about objects being blocks or tables should be held in preference to beliefs or assumptions about relationships between objects. For instance, when the *move(c,a,t)* e-element is applied to S_0 to generate S_2, there are other possible successors, each producible by selecting a different reconciliation set. The other possibilities are generated by the belief consistency maintenance function, when it suggests alternatives for belief revision. Given the results of *move(c,a,t)*, it will suggest that perhaps the assumption *block(c)* should be forsaken, since blocks cannot be on two things at once, and *c* is already on top of *a*. The heuristic we have suggested will not consider such reconciliation sets in the assumptive plan projection.

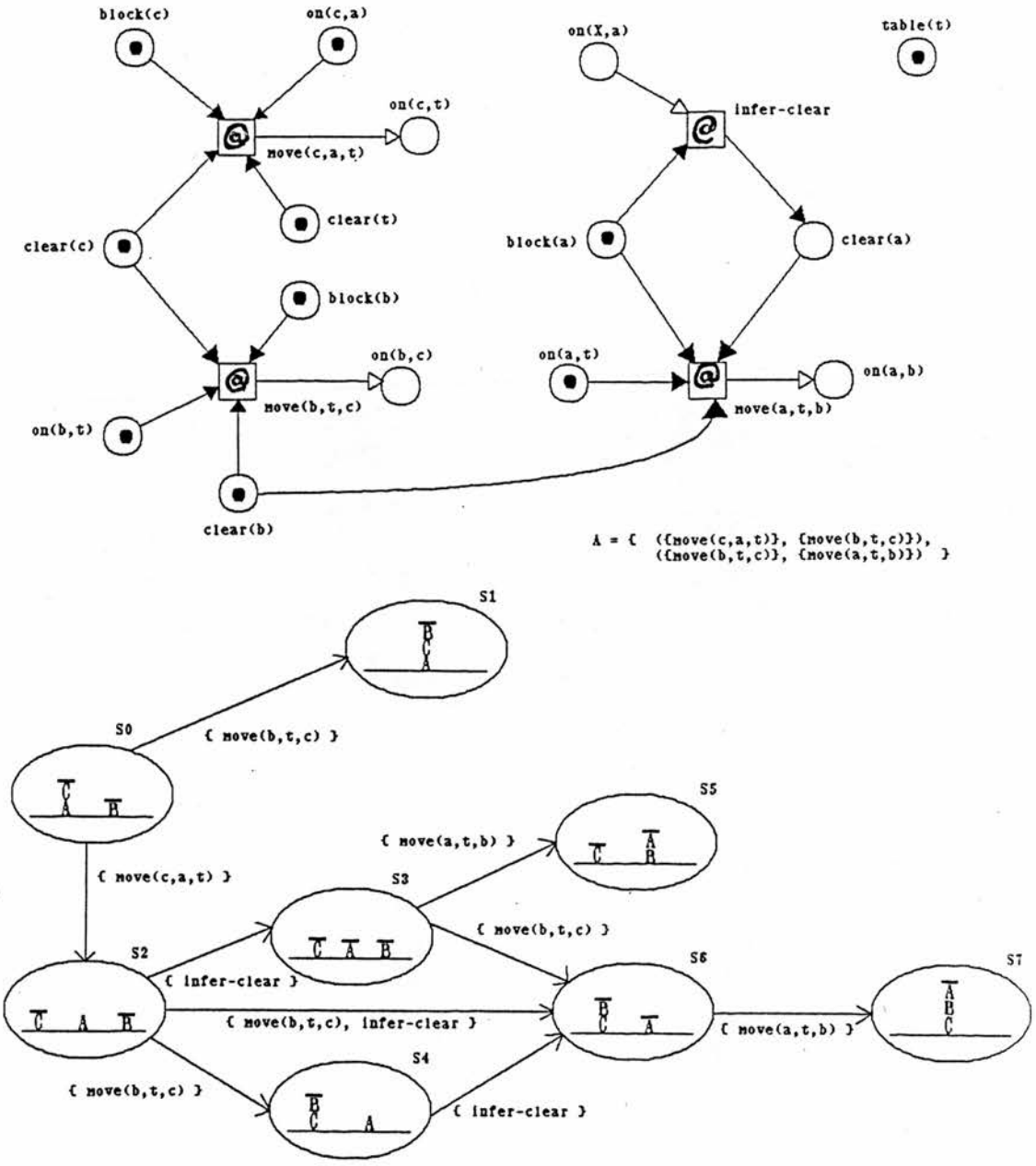


Figure 5.6: Stacking blocks.

5.7. Performing a litmus paper test.

This problem is borrowed from Moore (1984). The problem is designed to demonstrate the notion of a *test*; that is, a process by which the agent is to apply a procedure to determine an unobservable property of the environment by examining some other property which *is* observable. The idea for this test is that the agent wishes to come to believe something about the acidity of a given *solution*. This acidity is not directly observable, and so the agent must perform a litmus paper test: if the paper turns *red*, then the solution is *acidic*, and if the paper turns *blue*, then the solution is *basic*.

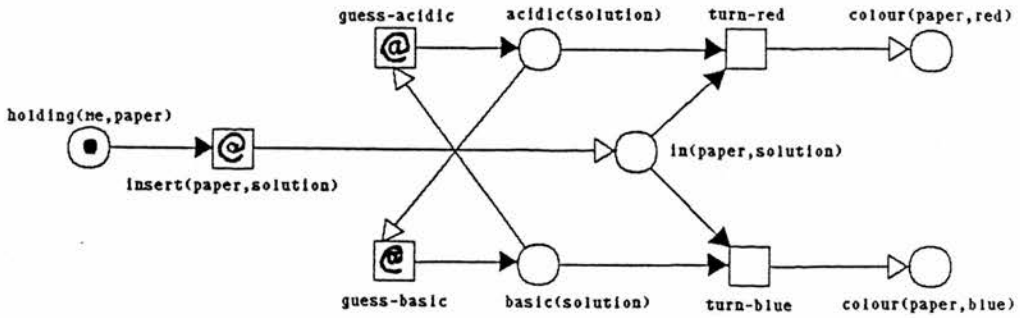
We formulate a solution to the problem as a plan net which includes the generation of two mutually incompatible beliefs. Two e-elements which model actions that generate beliefs are given in the plan net: one produces a belief that the solution is *acidic* on the basis that the formula *basic(solution)* is not *in*, and the other produces a belief that the solution is *basic* because the formula *acidic(solution)* is not *in*. These beliefs are used in the plan net's assumptive projection to see what might happen in either case; that is, what can be expected to happen if the solution is basic, and likewise, what can be expected to happen if the solution is instead acidic.

In our formulation of the problem, the agent initially assumes that it is *holding* the litmus *paper* required to perform the test. This assumption enables the action of inserting the *paper* into the *solution*. The only result of this action is that the paper will be in the solution. One of two events can then occur: if the solution is acidic, then in conjunction with the paper being in the solution, the action of the paper turning *red* is enabled, and will occur. Similarly for the other case, in which the solution is basic. In one situation, the plan terminates in S_6 with the assumption that the colour of the paper is red; and in the other, the plan terminates in S_7, with an assumption about the paper being blue.

Notice that either one of the two guesses and action of inserting the paper in the solution are causally independent, and can occur in any order. To save space in the projection, we have not labelled the arcs from S_0 to S_4, and from S_0 to S_5, with the obvious sets of e-elements.

A description of the plan might be as follows: "Insert the paper into the solution. If the solution happens to be acidic, then the paper will turn red; if instead, the solution is basic, then the paper will turn blue."

Notice that a domain constraint about any given solution being only basic or acidic was not required. This is because the guesses are self cancelling: if one occurs, the other cannot. In this net, there is no way that *acidic(solution)* can be made in when *basic(solution)* is already in, and vice versa. In general, a domain constraint specifying that solutions are not both basic and acidic would be required.



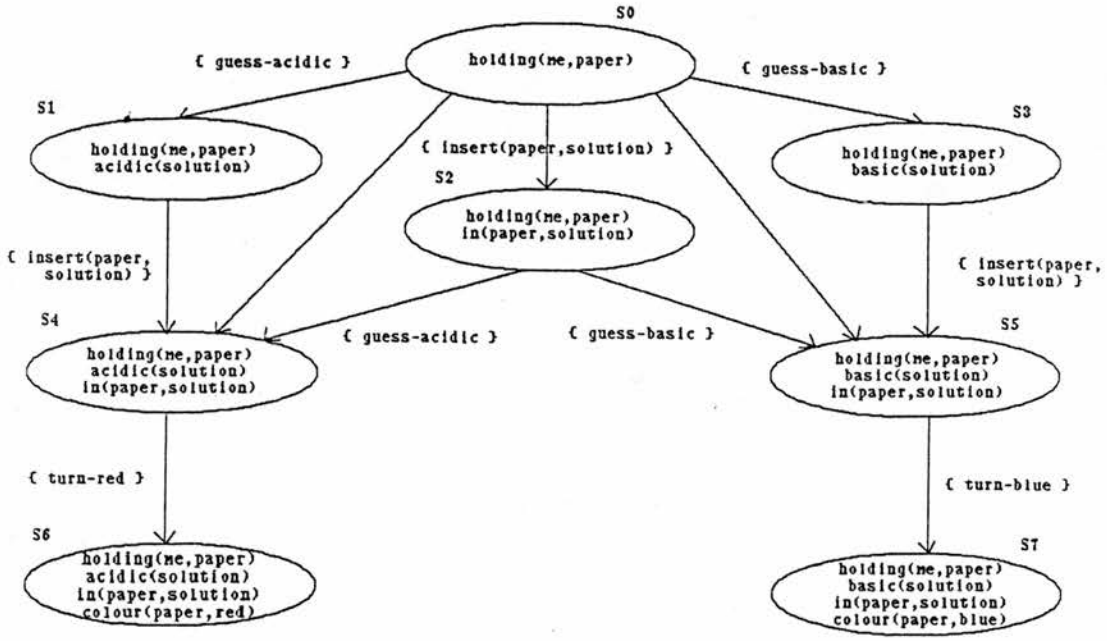


Figure 5.7: Performing a litmus paper test.

5.8. Planning to infer mortality.

The object of the plan given in figure 5.8 is to determine whether or not something called *frank* is a *mortal*. The agent is shown as having executancy for all events. The plan calls for counting the legs and arms of *frank*: the modelled counting actions have the fully-determined result of *frank* having two of each relevant appendage. If when actually counting *frank*'s, legs, it turns out that he doesn't have two, then the actions described by the given plan e-elements have not occurred as intended. The problem is one of mapping a potential solution into an actual solution. If plan execution fails, then this mapping fails; this does not mean that the plan is *incorrect* in any useful sense.

Following the counting of *frank*'s limbs, the preconditions for inferring that he is *human* are met. The action of inferring that he is human is modelled as causing a belief about his being *human*. The belief about *frank* being *human* permits the inference of his also being *mortal* to go through. The final result of the plan, as shown in S_6, is that *frank* is believed to have two legs, two arms, is believed to be *human*, and is also believed to be *mortal*. No domain constraints are required for this example.

This example raises an interesting question, first addressed in chapter 2. What is the difference between *inferential* and *sensory* actions? We suggested above that the action which results in a belief about *frank* being *mortal* is an "inference". The action of *counting frank*'s legs was advanced as being "sensory". What distinguishes the two e-elements to justify this difference in interpretation? Nothing. We might like to consider *sensory* actions to be those that are "information gathering"; however in general the classification of actions as information gathering, or not, will depend on where the information comes from (and perhaps also what the agent currently believes). An inference *can* be considered to be an information gathering action, since it definitely adds to the agent's store of beliefs about its environment. Somehow though, it seems that an inference only re-arranges "old" information, and that a sensory action actually provides "new" information. The only difference to be found between the actions occurs in the source of their information.

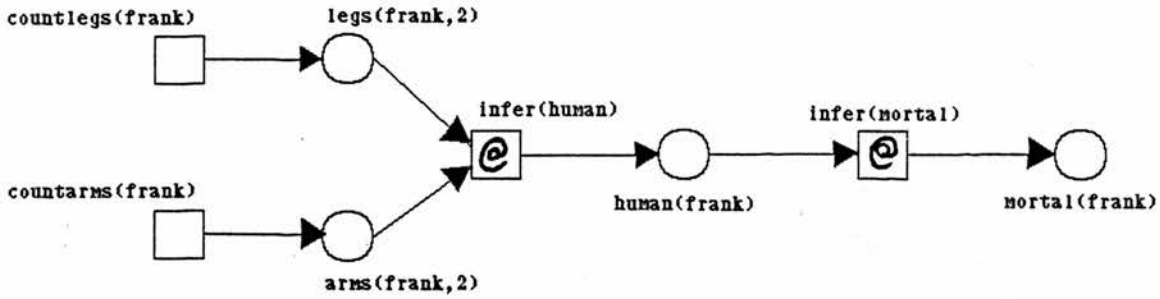
A true sensory action must use information from the environment in producing beliefs. It will use data from sensors to produce its internal results. An inferential action will use only existing world model data to add new beliefs. Both sorts of action are easy enough to implement, given the plan execution architecture presented in chapter 4. Given any e-element for which the @ address-generator is defined, the plan interpreter simply calls the program determined by the returned address. This program can do whatever it must to provide its results. It is free to access available sensors *and* the world model. When it returns its results, they will be integrated with existing world model formulae as shown in chapter 4.

The formalism we present makes no explicit distinction between inferential and sensory actions. A person implementing a system based on the formalism is free to make some actions inferential, and some sensory, as desired. However in terms of internal representation, both actions are described in exactly the same way. Whether an action is sensory or inferential, it will be described as having the result of causing a belief to be held by the agent.

There's a related point. It seems fair to suggest that different agents will be able to sense and infer different properties. An agent equipped with an infrared heat sensor would be able to directly observe whether or not a fully encased pilot light is burning. An agent restricted to observations using visible light will have no such option: it must enact a complicated pattern of actions, sensations, and inferences in order to determine the light's status (see example 5.12 below). Thus the actions that are sensory for an agent will depend on exactly how the agent is outfitted with sensors.

While we present no new insights on how to classify any given action as sensory or inferential, the formalism we define allows one to describe *either*. In both cases, the denoted action is characterised as causing new *beliefs* for the agent; the source of information for these beliefs is left unspecified. When providing the programs that implement the agent's low-level functions, a person is free to use whatever mechanisms are at their disposal to generate the results promised by the associated operators. A program can read a sensor, or it may simply re-arrange existing world model information; in either case, the results it generates must be of the general

form promised by the operator which describes the program's action.



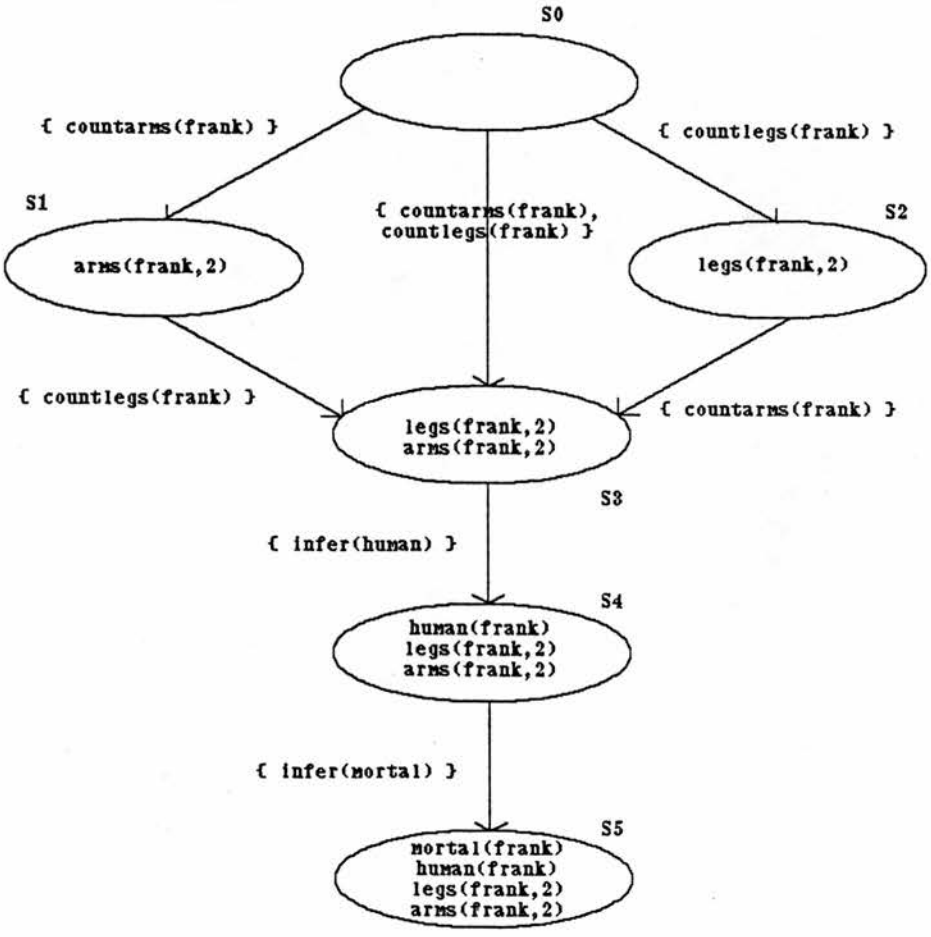


Figure 5.8: Planning to infer mortality.

5.9. Finding out the colour of the block.

This problem was first introduced in chapter 2, and involves an agent in coming to believe something about the colour of a specified block. In the current version of the problem, the agent initially assumes that its *eyes* are *closed*, that the *block* of concern is *at location1*, that the *lights* in its world are *on*, and that the agent itself is *at location2*.

As shown in the plan, there are three events for which the agent has executancy: opening its eyes, inferring adjacency, and going from *location2* to *location1*. The e-element *open-eyes* denotes the action turning on the agent's TV camera. The action by which the agent will actually come to believe that its eyes are open is not represented in this net. The e-element *adjacency* denotes the action of the agent inferring that two objects at the same location are also *next-to* each other. This action has the preconditions that the two objects of concern are believed or assumed to be at the same location. The final event for which the agent has executancy is represented in the plan net by the e-element *go(location2,location1)*. This denotes the action of the agent's going from *location2* to *location1*. The e-element's preset and postset seem intuitive.

The only event for which the agent does not have executancy is that of seeing the block's colour. The e-element *see-colour* describes this event. The agent believes that in order for the event to occur, its eyes must be open, the lights must be on, and it must be next to the block. Following occurrence of the event, a belief will be held regarding the *colour* of the *block*. The fact that the exact colour is not available in advance of plan execution is indicated by the variable *C* which is in the second argument position of the predicate *colour*.

It might seem as though attributing executancy to the environment for the event of the agent seeing the block's colour is rather arbitrary. In some ways, this is true. The e-element given in the plan is intended to describe an autonomous vision system, which operates without direct commands from the agent. Thus, the agent cannot direct this system in what to do. In this sense then, the agent does not have executancy over the event in which the vision sub-system provides a belief about a block's colour. If one wanted to model the action wherein the plan execution routine actually made a call to a vision sub-system, in order to do some analysis of the material

currently in the agent's visual field, then it would be better to say that the agent does have executancy over the action. But the question of executancy over this event does not change the way that it will affect the agent's beliefs: if the event denoted by the e-element *does* occur, then the agent will hold a belief about the colour of some block.

The domain constraints used are

{ {eyes(open), eyes(closed)}, {lights(on), lights(off)}, {at(me,X), at(me,Y)} }

and specify that the *eyes* are *open* or *closed*; that the *lights* are *on* or *off*; and that the agent is only *at* one location at any given time.

The projection contains a significant amount of parallelism, as can be seen by tracing from S_0 through S_3 to S_5, and from S_0 through S_1 to S_5. Once all preparatory actions have occurred, the projection shows that it is possible for the action of seeing the block's colour to proceed.

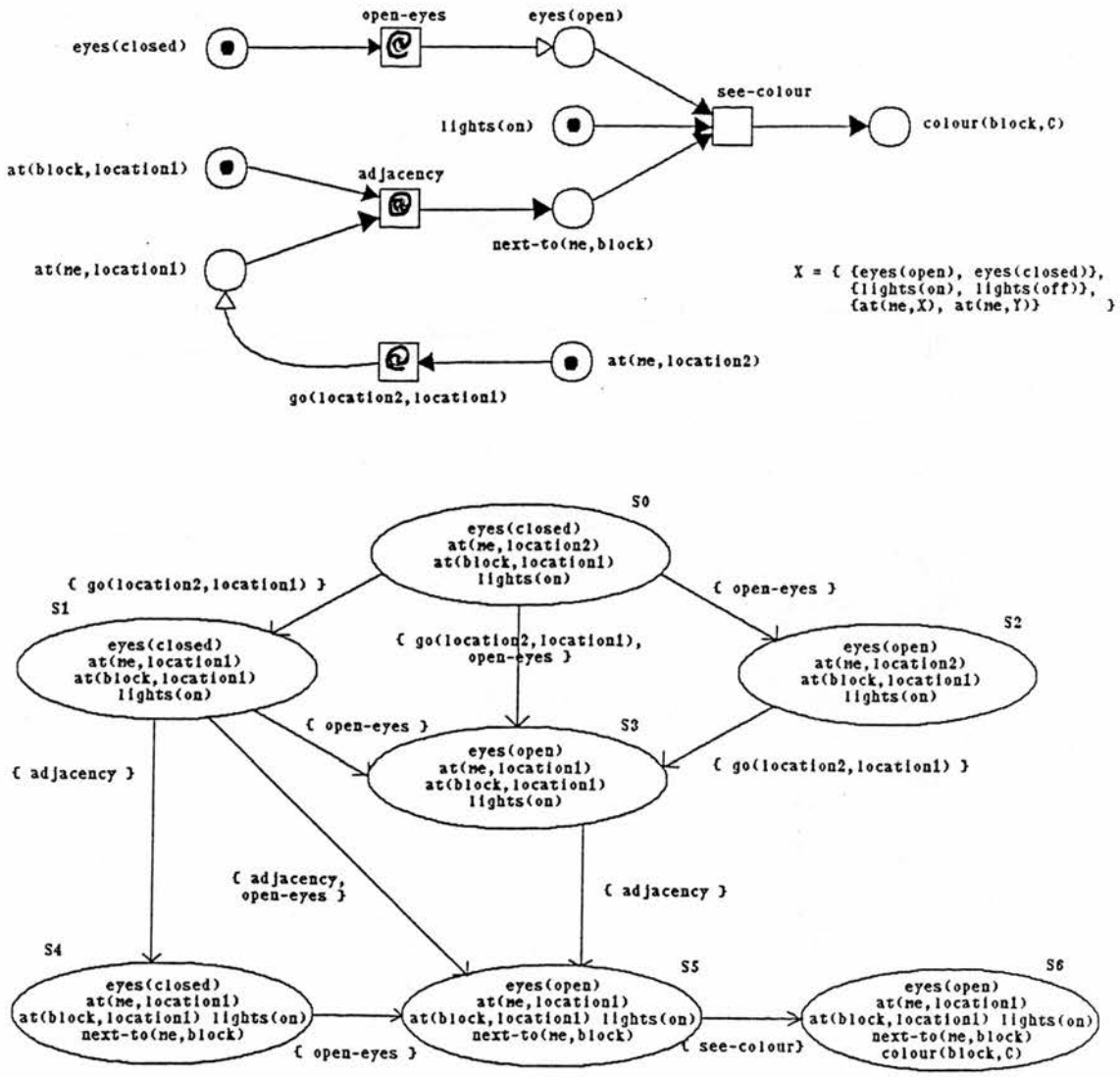


Figure 5.9: Finding out the colour of the block.

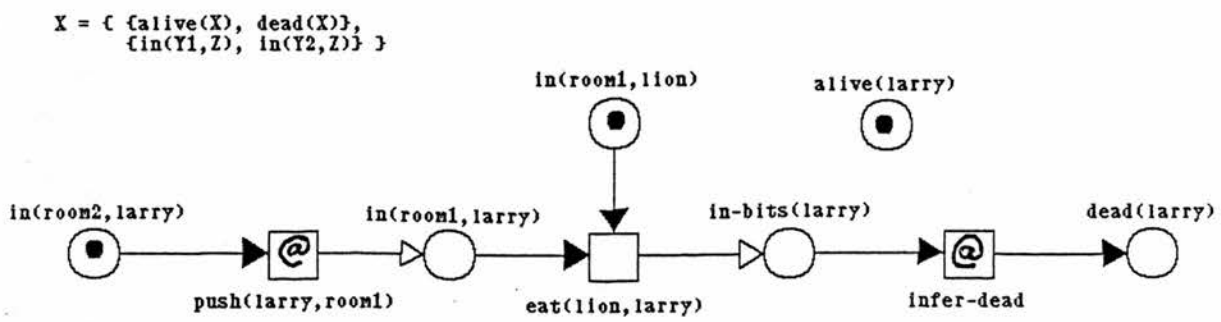
5.10. Planning to enable an external event.

Our next example is of rather dubious moral standing. The plan net calls for *larry* to be eaten by a lion. This example illustrates that from the perspective of the agent generating and reasoning about the plan, event executancy does not matter. In this plan there is an event of the *lion* eating *larry*: the agent cannot be said to have executancy for the event, but can reason about how the event will affect the environment and its own mental state.

Initially, the agent assumes that *larry* is in *room2*, that *larry* is *alive*, and that the *lion* is in *room1*. The stage is set for *push*ing *larry* from *room2* into *room1*. Given that *larry* and the *lion* share a room, the *lion* will *eat larry*, resulting in *larry* being *in-bits*. The condition of *larry* being *in-bits* permits the inference of *larry* being *dead* to go through.

The domain constraints required for this example specify that nothing is both dead and alive: {alive(X), dead(X)}; and that things are in at most one room: {in(Y1,Z), in(Y2,Z)}. The first constraint is used to force out the assumption regarding *larry* being *alive* during the transition from S_2 to S_3. In S_3, *larry* is believed to be *dead*; the belief being sanctioned by the inference from his being in bits. The formula describing *larry*'s being in bits is an assumption (in S_2 and S_3).

And as we keep insisting, event executancy doesn't enter into plan net projection. This means that it is quite easy for the agent to model the occurrence of events completely outwith its control.



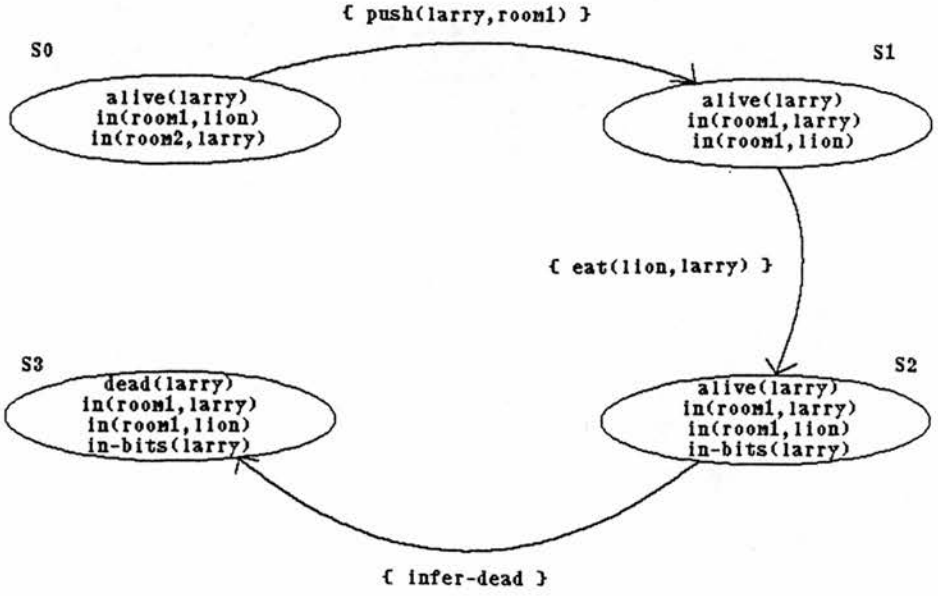


Figure 5.10: Planning to enable an external event.

5.11. Dialing a combination safe.

In this example, the agent is required to open a combination safe, and does not initially have the necessary combination. However, the combination of the safe is written on a scrap of paper which the agent has in its possession. The idea is that if the agent reads the combination from the paper, it will then have all the information required to open the safe.

We formulate the example using two e-elements: one to model the action of reading the value of the combination from the paper, and one to model the action of dialing the combination on the safe. The preconditions for the action of *read*ing the paper are that the agent believes or assumes itself to be *looking-at* the *paper*, that the agent believes or assumes that the *combination* is *written-on* the *paper*, and that the agent believes or assumes that the *lights* are *on*. If all these conditions are met, the action of reading the paper can occur. In the initial case given in S_0, these preconditions are met. The *read* e-element models the action of reading the combination as having an internal result of the *combination* having a certain *value*, *Y*. The variable *Y* is used to indicate that the *value* of the *combination* cannot be determined in advance of plan execution. So the action of reading the *combination* causes a belief for the agent about the value of the safe's combination. This belief, plus the assumption that this *combination* in fact *opens* the specified *safe*, enables the e-element which models dialing the combination on the safe. Dialing the combination on the safe is described as having the result of the *safe* being *open*. This is used to generate an assumption in S_2 regarding the *safe* being *open*.

The *dial* e-element is another example of an action specification that is not fully complete when plan execution begins. In order to constitute an executable description of the action of dialing a combination on a safe, it must have all of its variable parameters bound. If the action modelled by the *read* e-element does not actually occur, then no belief about the *value* of the safe's combination will be held when the time comes to open the safe. All this means is that the @ function will not be able to give an address for the *dial(Y,safe)* e-element. Alternatively, it is possible to consider that the @ function *is* defined, but when the routine is called, it cannot do what is intended of it, and fails, due to lack of information.

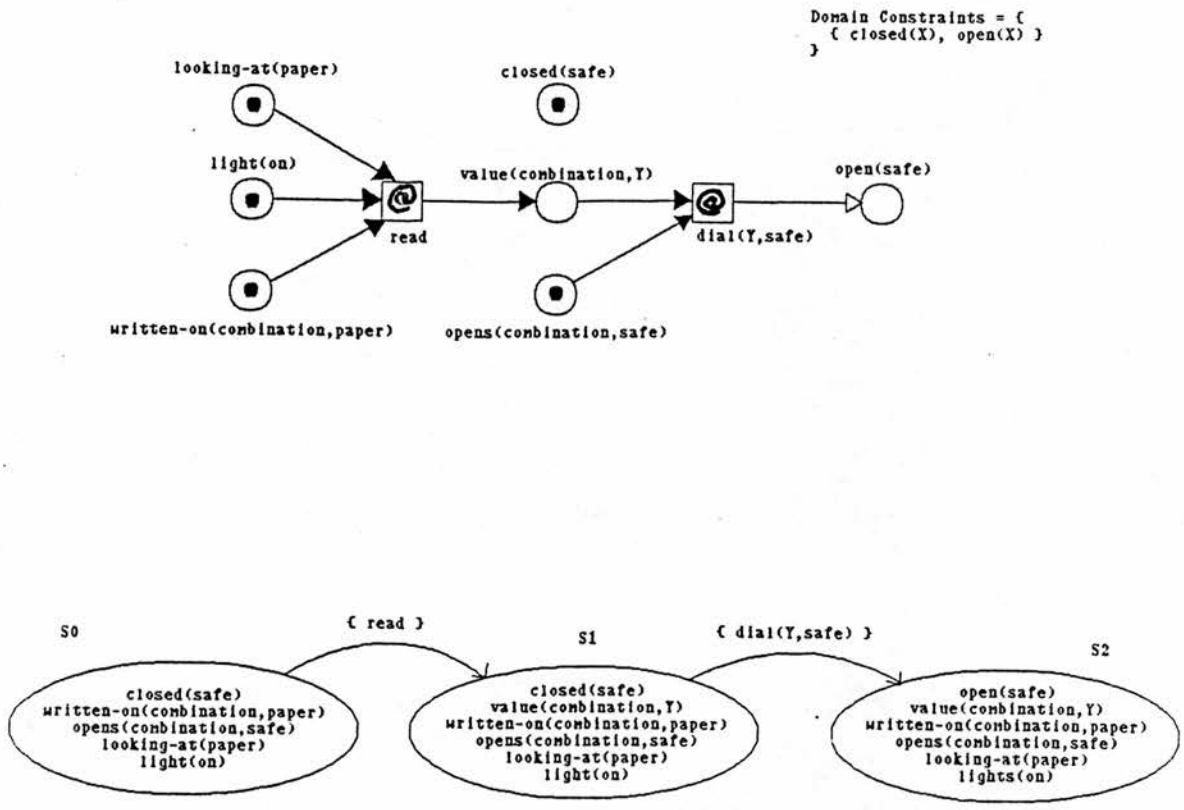


Figure 5.11: Dialing a combination safe.

5.12. Checking the pilot light.

The problem used to create this example comes originally from Konolige (1980). We phrase the problem slightly differently, and give a *plan* by which the agent can come to solve the problem. The problem is this: there is a fully encased, and thus unobservable, pilot light on a gas stove. The agent's goal is to determine whether or not the pilot light is burning. The plan we present for the agent to solve the problem can be paraphrased as follows: "Perhaps (since I have no information to the contrary), the pilot light is on. Turn on the gas to the burner. If the pilot *is* on, then the burner will ignite, confirming this guess about the pilot light. If the burner does not ignite, and the gas is flowing, then the pilot light must be off. Either way, at the end of this procedure, I'll come to believe either that the pilot light is on, or that the pilot light is off."

There are two domain constraints required:

{ {burner(X1), burner(X2)}, {pilot(Y1), pilot(Y2)} },

which specify that burners cannot be in two states at once (in the example, *on* or *off*); and that pilot lights are also never in more than one state (also *on* or *off*). These constraints are used to recover from faulty guesses and to change belief in the status of the burner, as it goes from *off* to *on*.

The only initial assumption provided is that the *burner* is *off*. This is shown in S_0, where two causally independent e-elements have concession: *turn-gas-on* and *guess*. These model the actions of turning the gas supply on, and guessing that the pilot light is on. Since the actions are modelled as being causally independent, they can occur in any order, or in parallel. Turning the gas on results in a possible state of belief shown in S_1. In this state, the results of turning on the gas have been added as an assumption, and the assumption forces nothing else out. From S_1, one of two disjunctive actions can occur. Either, the agent can infer (modelled by the e-element *infer1*) that the *pilot* light is *off*, or *guess* that the *pilot* light is *on*. The *infer1* e-element causes a transition into S_4, where the agent believes that the *pilot* light is off, assumes that the *gas* is *on*, and that the *burner* is *off*. No further transitions are possible from this case. The formulas *gas(on)* and *burner(off)* are assumptions, since they cannot be discharged by any e-element given

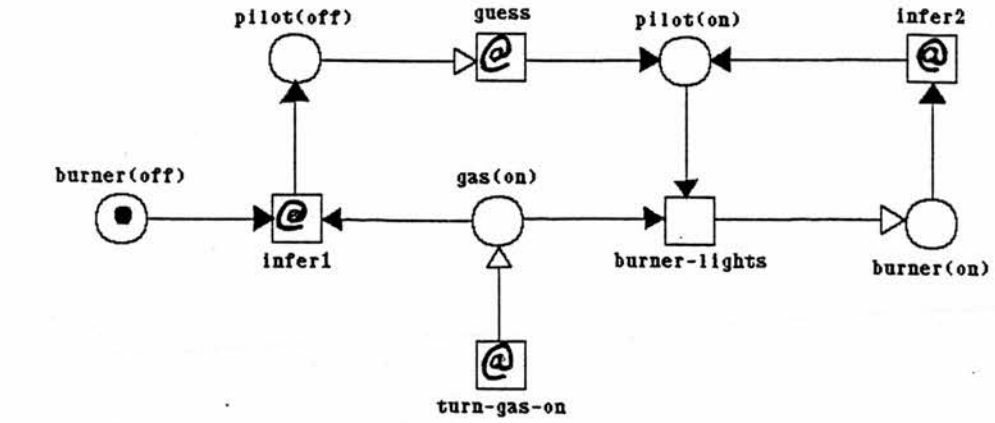
in the plan net. The e-element *pilot(off)* is a reasoned belief, since it is justified by the inference modelled by *infer1*, and can thus be discharged by it.

In S_3, the agent believes that the *pilot* is *on*, and assumes that the *burner* is *off* and that the *gas* is *on*. Once again, from this state, there are two possible events: the agent can infer (modelled by the e-element *infer1*) that the *pilot* is *off*, or the *burner* could *light*. If the agent infers that the *pilot* is *off*, then by the definition of assumptive transition, those existing formulae incompatible with those being added must be removed. In this way, the formula *pilot(on)* is removed from S_3 in order to generate S_4. The behaviour that runs from S_0 through S_1 and S_3 to arrive at S_4 models the agent turning the gas on, guessing that the pilot light is on, and then inferring that, after all, the pilot light is not on. The behaviour that runs from S_0 through either of S_1 or S_2 to S_3 and then to S_5 models the agent guessing that the pilot is on, and turning the gas on (in some order), and then having the burner light. The transition from S_3 to S_5 forces out the assertion *burner(off)*, which was given as an assumption in the initial case. This is because the lighting of the burner makes it impossible keep *in* the assumption that the burner is off (by the given domain constraints).

No matter which path is taken through the projection, there are only two terminal cases. In S_4, the *burner* is assumed to be *off*, and the *pilot* light is believed to be *off*. In S_5, the *burner* is assumed to be *on*, and the *pilot* is believed to be *on*. In both cases, the *gas* is assumed to be *on*. The formula *pilot(off)* in S_4 is believed with reason *infer1*.

It is possible to give two different reasons for the e-element *pilot(on)* to be in S_5: either the *guess* that it is *on* (because it may not be *off*), or the inference (modelled by *infer2*) that it is *on* because the *burner* is *on*. Notice that the definition of belief we have given thus permits the justification of a b-element by an e-element which was not actually used to produce it. An extended justification for a b-element being in will take the form of a recursive justification: the basic preset of the e-element used as a reason for the b-element being in must be justified. The enable-in preset must be shown to have reason to be in, and the enable-out preset must be shown to have reason to be out. The definitions we have given look only one level back into a possible

plan net firing to determine the status of a b-element. Other definitions are possible which we have not considered.



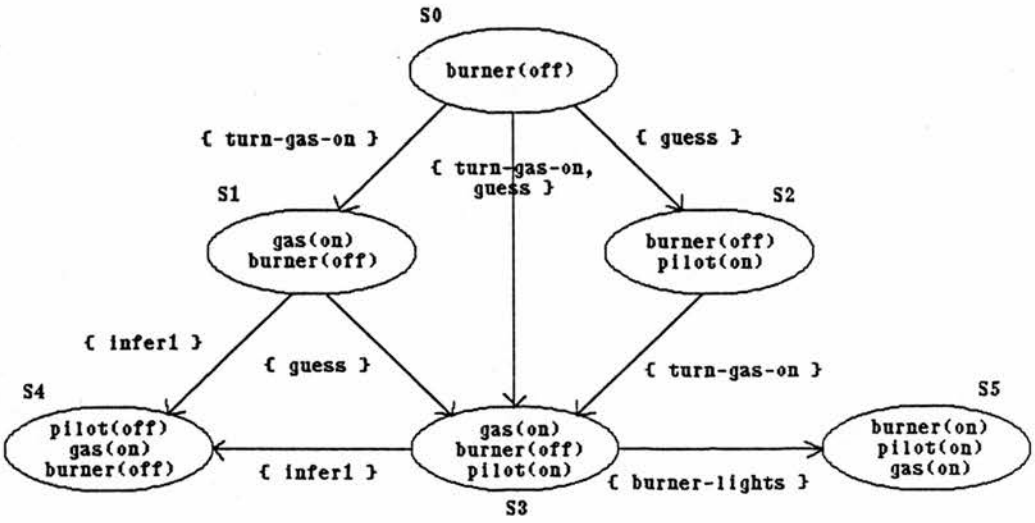


Figure 5.12: Checking the pilot light.

Chapter 6: Discussion.

6.1. Chapter overview.

The chapter follows up the technical definitions of chapter 4 and the examples of chapter 5 with some discussion of the plan net formalism in terms of related representations, theories, and general issues. We address the relationship between plan nets and STRIPS operators, reason maintenance systems, and modern representations for teleological information. The issue of plan net generation is briefly explored.

In section 6.2, we return to STRIPS, and compare its operator representation with what we have developed. The relationship between STRIPS' triangle tables and the teleological information used to define ordering relations in plan nets is addressed. We also present a relationship between the STRIPS system search-space and a plan net's assumptive projection.

In section 6.3, the relationship between Sacerdoti's (1975a, 1975b) procedural net and the plan net is discussed more fully than was possible in chapter 3.

While the plan net formalism may not appear to directly address the problems dealt with by conventional reason maintenance systems, it seems that much of the required functionality is indeed present. Section 6.4 points out the relationship between the plan net and Doyle's RMS (1979, 1983, 1985) work, and de Kleer's (1984, 1986a, 1986b, 1986c) assumption-based approach to the problem of reason maintenance.

It has been suggested that plan nets contain a great deal of teleological information. This claim is borne out in section 6.5, by detailing the relationship between the information used to construct plan nets, and other versions of this information available in current-day planners. Tate (1984b) calls such information "goal structure", and Wilkins (1984a) calls it "plan rationale". We discuss both these concepts in terms of the plan net definition.

A few ideas on plan net generation are advanced in section 6.6. We present a very simple method by which plan nets may be generated, and comment on how the concepts and definitions of chapter 4 allow a planner to cache previously derived results, resulting in more efficient plan generation.

One overall comment on the examples used in this chapter: often, the language L_B is extended without explanation. These extensions facilitate comparison of the plan net framework and the system under discussion, and pose no problems. Where problems might be caused to the definitions of chapter 4, they are noted, and alternative formulations suggested.

The chapter has no conclusion, since each section is essentially a stand-alone commentary, and thus contains its own conclusions.

6.2. Comparing STRIPS operators and plan nets.

Using the definitions of chapter 4, we can now discuss more fully the advantages enjoyed by plan nets over STRIPS (Fikes & Nilsson, 1971) operators. As in chapter 2, recall that while the discussion is in terms of the STRIPS system, all major planning systems to date have copied the STRIPS representation for operators. We thus refer to "STRIPS-form" operators, and by this term mean any representation based on the underlying add-list and delete-list idea. In the discussion below, we begin with an expanded commentary on the idea of add-lists and delete-lists, and then move on to show some correspondences between STRIPS operator application and plan net projection. We finish with some ideas on the relationship between STRIPS triangle tables and plan nets.

6.2.1. Basic operator representation.

Consider the action which has appeared in a variety of places in the previous chapters: an agent moves some block *A* from *B* to *C*. During the course of the action, a weight sensor in the agent's arm supplies information regarding the weight of block *A*. The notional effect of the action is to make block *A* be on top of block *C*. How might this action with all its results be represented to the agent via the STRIPS operator formalism? This question can be answered in two parts, one answer for each of the add-list and delete-list.

What would the add-list be? Should this list contain an assertion of the form *weight(a,X)*, and an assertion of the form *on(a,c)*? In different situations, the agent will want to achieve

different results -- packing these two formulas into a single add-list ensures that the agent will not be able to reference the action for the required results. Some of the action's effects are *physical*, out-there-in-the-world results, results which involve blocks actually being on top of each other. Other of the action's effects are *mental*; that is, inside-the-agent's-head results, which result in the agent coming to believe something about the world. These results are quite different, and the operator representation language must distinguish between them if the agent is to be expected to invoke actions for the appropriate results. The plan net formalism explicitly provides the language for making such a distinction, and STRIPS-form operators do not.

What should the delete-list for the *move* action be? First, what is the delete-list intended to describe? The operator of which it is a part is intended to describe an action. Following operator application, the formulas in the delete-list will be removed to derive a successor "model" to the one in which the operator is applied. Thus, the delete-list would appear to describe the beliefs that the action modelled by the operator would remove from the world model, were it to actually occur. This seems rather suspect. It appears strange to say that performance of an action forces formulas out of the enabling model. Rather, we might want to say that the actual additions to the formulas of the agent's model under the simulated occurrence of the action is what would cause existing formulas to be rejected. However, this suggests that those formulas to be rejected depend on what is actually added, and perhaps, what is already in the enabling model.

Consider the following delete-list scenario. An operator somehow calls for the addition of a formula *on(a,b)*. What should be deleted? Clearly, this depends *only* on the agent's current beliefs regarding *a* and *b*. What if *b* is believed to be a table? Then it would be fine for the agent to believe as well that *b* is clear. However, what if *b* is believed to be a block? (Being a block implies that *b* is capable only of unit-area support.) Then it would seem impossible for the agent to believe as well that *b* is clear, since *a* is on it. In addition, consider the object denoted by *a*. What beliefs regarding it should be rejected? If it is a block, then *b* must be the only object is is on; if not, then it may well be on *b and* some other object(s) simultaneously. The situation can be summarised as follows. The beliefs that must be rejected under operator application are only a function of what is currently believed by the agent, and what is being

"added" by the operator.

There is another way of looking at this delete-list problem. STRIPS-form operators commit themselves to dealing with objects of a specific *type*. In all STRIPS-form approaches to the classic blocks-world domain, four different operators are required. One operator describes the action of picking up a *block* from the *table*; one describes picking up a *block* from another *block*; one describes placing a *block* on the *table*; and one describes the action of placing a *block* on another *block*. Why are four operators required? Simply because the operators' delete-lists commit them to dealing with a particular sort of object. For example, the operator which describes the action of placing one block *X* on another block *Y* specifies in its delete-list that following the occurrence of the denoted action, *Y* will not be clear. In contrast, the operator which describes the action of placing some block *X* on the *table* does not say that the *table* will not be clear following the action's occurrence. The belief consistency method incorporated in the plan net projection mechanism provides a means for deleting beliefs without attaching delete-list specifications to each operator. This means that in the blocks world, we require only one block movement operator: move block *X* from some object *Z* to some other object *Y*. (In fact, this operator is the one used as an example throughout chapter 4). The deletions forced by the operator are determined from the context and method of its application.

6.2.2. Methods of operator application.

Given that we make a distinction between the internal-results and external-results of an action, we must also make a decision about how the sets which describe these results in an operator are to be used in operator application. This choice has been formalised in chapter 4 as two sorts of projections, each one depending on a different method of operator application: assumptive or realistic. It seems reasonable to ask about the relationship between these two sorts of projections and the STRIPS system method of operator application.

First, since STRIPS does not provide for distinguishing between the internal and external results of an action, it cannot support the two different methods of operator application. However, with this restriction in mind, we can consider which of the operator application methods we have

defined is closest to that performed by STRIPS. Since STRIPS does use the declared external results of actions (as specified by the operator add-lists) to derive successor models, it appears closest to our *assumptive* operator application. Successor models are produced by *assuming* that actions will function as specified by their operators. However, that is the closest analogy that one can draw: without a distinction between internal and external results, it is impossible to characterise beliefs and assumptions appropriately.

One other point is worth making here. Recall the discussion in chapter 3 on plan representation and construction: following recent comments by Charniak and McDermott (1985) and Tate (1985), it was shown how plan construction can call for search through world states or partial plans. STRIPS was a system that searched through world states. (More correctly, we would now say through belief states, but we can let the terminology slip slightly.) This sort of state-space plan construction reasoning has the advantage that it is extremely easy to perform question-answering operations, but limiting in that plan construction is driven by simulated behaviour enactment (the plan construction reasoning and search-space navigation procedure are the same). A system based on the plan net representation can use one of the two forms of projection to do its question answering, but has the advantage that it can still search through a space of partially completed plan nets, if desired. At each stage of the search, a plan net projection can be carried out to support question-answering.

6.2.3. Plan net as triangle table.

One of the interesting features of the STRIPS system was its basic representation for plans. A completed plan was represented as a *triangle table*. This table recorded all the information required by the plan execution system, Planex. The underlying idea was that the triangle table would record all the dependencies between planned actions in a way that made monitoring the execution of the plan simple. In particular, it was desired that action failures be detected as soon as possible, and that opportunistic action execution be an option, should action preconditions become independently achieved. Essentially, the triangle table expressed information regarding plan *teleology*; that is, information about action purpose and action inter-dependencies.

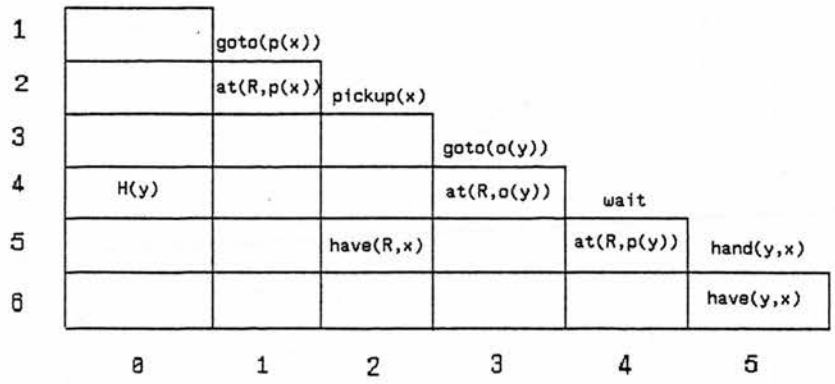


Figure 6.1: The triangle table *deliver(y,x)*.

Any given triangle table can be easily converted into a plan net, under the restrictions imposed by our changed understanding of the functionality of add-lists and delete-lists. Thus the teleological information given by any triangle table can be easily put into plan net form. For a simple example, we borrow from Nilsson (1985, p.6). Nilsson's original triangle table is given in figure 6.1, and a plan net version of the same teleological information is given in figure 6.2. To be able to formally represent the net shown in figure 6.2, is would be necessary to extend the languages L_B and L_E, to allow formulae to contain functional expressions. In practise, this isn't necessary, since such expressions can be "flattened out", as the examples of chapter 5 should serve to illustrate. However, for the purposes of this example, we remain as close as possible to the basic structure of the formulae contained figure 6.1.

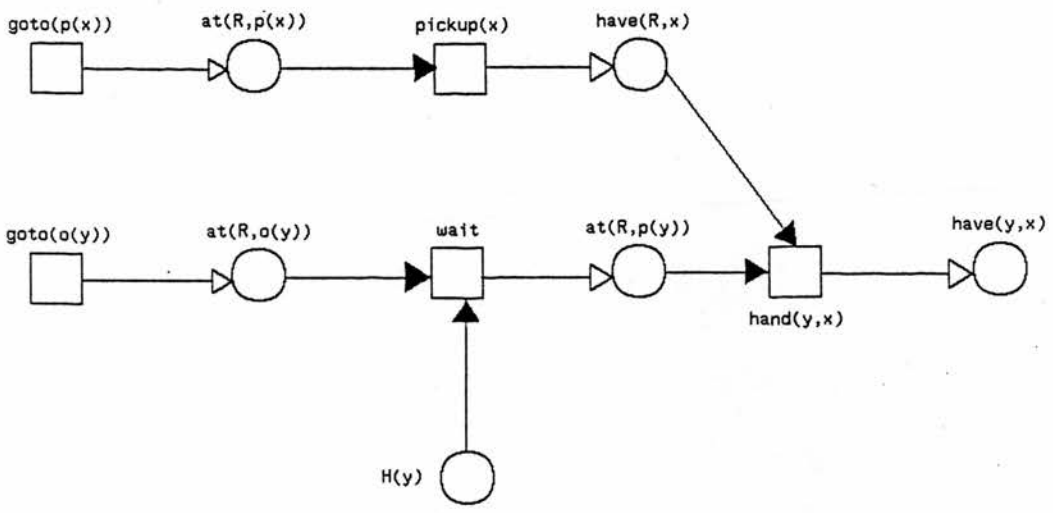


Figure 6.2: A possible plan net for *deliver(y,x)*.

The intended interpretations for the predicates and functions are as follows. First, x and y are variables, R is a constant. *at(R,x)* means the agent R is at place x. *H(y)* means that the agent y is at work today. *have(y,x)* means agent y has object x. *p(x)* evaluates to the present location of object or agent $x1$. *o(y)* evaluates to the usual location of agent y. R denotes the agent itself. Under this interpretation, the plan calls for the agent to go to the object's location, pick the object up, go to the office where the recipient usually is, wait for the recipient (if necessary), and then hand over the object.

Notice that triangle tables do provide one bit of information not *explicitly* included in a plan net. This is information regarding formula persistence across action occurrence. In a triangle

table, formula persistence is indicated by last listing the formula in the column opposite the action which deletes it. In order to include this information in the plan net framework, we must define the appropriate domain constraints. For this simple example, the domain constraints required should be obvious.

6.3. Relation to Procedural nets.

Plan nets are, in a very rudimentary way, similar to Sacerdoti's (1975a, 1975b) procedural nets. The two representations share an action-ordering philosophy: a plan is built from actions and ordering relations. But this underlying action-ordering philosophy is just about all the two representations share. The procedural net was the inspiration for much of the work presented in this dissertation. But as with many endeavours, the end result is quite different from its inspiration. The following paragraphs briefly discuss the major differences and abilities of the two representations. As in chapter 3, by the term "procedural net", we mean the entire class of action-ordering plan representations which derive from it.

The order in a procedural net is a strict partial order: irreflexive and transitive, thus asymmetric. This is intuitively sound, since the ordering is intended to reflect the "flow" of time. If one action α is *before* another action β, then β cannot be before α. No loops can ever exist in time, so no cycles can exist in a procedural net. This poses a problem when attempting to model iterative behaviours of agents; for instance, agents can hammer nails, can walk until they come to a bar, and can dial a number on a telephone until someone answers. It agents can take part in such activities, it seems natural to suggest that their representation for plans be able to capture the required notions. We would not want to attempt a solution to the problem by introducing "special" nodes into a net which describe iterative actions. Such a ploy would hide the notion of process inside an inaccessible black box: if iteration is to be represented, it must be represented in such a way that permits reasoning about it. This is not to say that we would not want the ability to abstract away from iterative constructs, and view them as single-step actions. It is important however, to be *able* to represent iterative constructs in a net in much the same way that linear ones are. In this thesis, we have adopted the philosophy that in order to describe iterative

constructs, a richer ordering structure is required.

By using the basic presets and postsets of e-elements, the construction of iterative plan nets is straightforward. The interpretation of the ordering relation on b-elements and e-elements in a plan net is not that of *before*. In terms of the suggested interpretation, we can consider the holding or not holding of certain conditions to *enable* the occurrence of an action. This is represented to the agent by an e-element (denoting the event) which stands in specified relations to two sets of b-elements. One of these sets must be in (denoting those conditions which *do* hold), and one set must be out (denoting those conditions which do *not* hold). The occurrence of an event can cause the holding of beliefs by the agent and holding of conditions in the environment. The relation between an event and its results is characterised for the agent by two relations on e-elements and b-elements. We can consider both relations to characterise the general notion of *causation*; thus, events cause the holding of conditions and beliefs.

There is no requirement that these "cause" and "enable" ordering relations have no loops, as the examples of chapter 5 should clearly demonstrate. The basic difference between a plan net and procedural net seems to be that a procedural net is attempting to model time *directly*; a plan net says nothing about time – in the plan net framework, time only appears as something that is required for a plan net to undergo change. This reflects the basic philosophy that an agent need not represent time explicitly: all that the agent is required to represent is the potential for *change*. Using a plan net, time is only that which is required for change.

Defining this order on actions and beliefs also buys us an explicit representation for a plan's teleology, as mentioned above in the context of STRIPS' triangle tables. In the procedural net, such information was only implicit, and surfaced in the diagrams used to present the results of the NOAH planner. These annotations did not form part of the underlying plan. One bit of the information did appear in a structure called the Table of Multiple Effects, or TOME. The TOME was used to collect together all action effects produced in one expand-criticise cycle. We use such information to *define* a plan net.

Sacerdoti did not comment on basic plan analysis. Indeed, one of the suggestions in his dissertation is that the plan modifications performed by NOAH's critics are rather *ad hoc*, and could easily stand up to more rigorous study. (This suggestion has been well followed up by Chapman, 1985). We have presented plan net projection as a tool for net analysis. Using the definition of net projection, it is possible to directly write a computer program which can perform most required question-answering functions.

Related to the issue of iteration is the inability of the procedural net to cleanly describe disjunction of action. Now the obvious solution is to introduce another "special" node which describes some notional *or*-action. In the plan net framework, by using the definition of e-element occurrence, it is easy to build a plan net which contains disjunctive actions. Two e-elements can easily be enabled in any given case, and following the occurrence of one, the other can be disabled. Once again, this definition is made possible only through the underlying ordering relations on b-elements and e-elements.

Finally, note that all the comments made above regarding STRIPS-form operators still hold for the procedural net, since operators basically describe actions by using simple add-lists and delete-lists.

6.4. Relation to Reason maintenance systems.

It turns out that much of what is required of a Reason Maintenance system is provided by the plan net formalism. This section explains the basic relationship between plan nets and two of the major approaches to the reason maintenance problem: Doyle (1979, 1982, 1983, 1985) and de Kleer (1984, 1986a, 1986b, 1986c).

6.4.1. Basic ideas.

A reason maintenance system is charged with keeping track of current "beliefs", given a set of justifications and assumptions. The various flavours of system available can be derived by playing with the exact definition of *belief*, *justification*, and *assumption*. The two most successful

approaches to the problem are Doyle's (1979, 1982, 1983, 1985), and de Kleer's (1984, 1986a, 1986b, 1986c).

The basic idea is that a computer reasoning program constructs computational models of the world. As new information is made available to the program, it will need to revise its model. Since some parts of the model are constructed from other parts, there will be many *implicit* dependencies present between the atomic assertions or formulae from which the parts of the model are built. Changes in one part of the model will have repercussions in other parts. In order to keep track of which parts of a model must change, *explicit* dependencies between the basic formulas must be maintained. In the rest of this chapter, we refer to any system which is charged with this sort of functionality as an RMS. Doyle's (1979) original system was called TMS, for Truth Maintenance System, but the title was dropped, since what it does has very little to do with truth. The current invocation of Doyle's system is called RMS, and it is this system that we take as a typical instantiation of the entire justification-based reason-maintenance paradigm.

The two fundamental operations which an RMS must perform are generally called *truth maintenance*, and *dependency-directed backtracking*. The first term is a hold-over from Doyle's (1979) original TMS, but we retain it to be compatible. Truth maintenance refers to the process of determining what must be believed, given the set of current justifications. Dependency directed backtracking is the process of fixing contradictions which occur in the belief system by finding the basis for the contradiction, and changing something in this basis so as to avoid the contradiction in the future. de Kleer's (1984) original Assumption-based Truth Maintenance System (or ATMS) addresses only the problem of truth maintenance, but recent work (de Kleer & Williams, 1986) has begun to consider how dependency directed backtracking can be fitted into the ATMS framework. In the sub-sections below, we show how the plan net formalism and technique of plan net projection incorporate the truth maintenance process; we also demonstrate that plan net projection includes at least *part* of what is required of dependency directed backtracking.

While reading this section, remember that an RMS is intended to be a generally useful subsystem for problem solvers. We have defined plan nets directly in terms of the languages L_B

and L_E, and have thus made a rather serious commitment to a particular mode of representation. This is fair, since plan nets are intended to be the underlying representation of a *planning* system. An RMS talks only in terms of "nodes" and "justifications", and makes no commitment as to how these objects will be manipulated by the reasoning component. Normal RMS representations are *not* however adequate as a representation for *plans*, since they do not incorporate the required notions of *change* and *time*. Fundamentally, describing change over time is what plan nets are all about.

6.4.2. The basic correspondence.

Doyle's (1979) early work used informal data-structure and algorithm presentations of the basic RMS ideas. Later Doyle (1982) formalised the work, using set theory. Doyle (1983) presents the major results of Doyle (1982), and in 1985, he extends the framework by showing how simple reasons can be interpreted as economic notions. Unfortunately, some change in terminology has occurred. In the early work, justifications (the reasons for a belief) and a node (which represents the belief to the system) were distinguished from each other (see for instance, Doyle, 1979, p.236). In the more formal presentation (Doyle, 1982, 1983), these two objects are not so cleanly separated. As will become clear in what follows, in the plan net formalisation, we *must* distinguish between the two.

For Doyle (1979), a support-list justification is a pair of sets of nodes. The first set is called the *in*list, and the second is called the *out*list. The justification is *valid* if and only if all the nodes in the *in*list are *in*, and all the nodes in the *out*list are out. A node is *in* if and only if its justification is *valid*. In Doyle (1982, 1983, 1985), this idea is made more precise, and is generally notated $A \,||\, B \,||\!\vdash C$. This is read "$A$ without B gives C", and is a more concise notation for the earlier support-list justification. A is the set of nodes that must be *in*, B the set of nodes that must be *out*, and C the set of nodes supported by the justification. In the new terminology, these justifications are called *simple reasons*. Given a simple reason r, its three component sets can be referenced as $A(r)$, $B(r)$, and $C(r)$. In what follows, we use the terms *simple reason* and *justification* interchangeably.

The basic correspondence between a simple reason and a plan net is as follows. We can interpret a plan net $P = (B,E;F,A)$, as a collection of simple reasons; each simple reason is an e-element $e \in E_P$. Given one of Doyle's simple reasons $r = A \,||\, B \,||\!\vdash C$, and an e-element $e \in E_P$, the basic correspondences are given by:

$A(r) = {}^{+}e,$

$B(r) = {}^{-}e,$

$C(r) = e^{+}.$

Notice that there is no RMS construct equivalent to a plan net e-element's external-results postset, e^{-}. In what follows, we will pretend that this set ceases to exist, in order to make the comparison simpler. It makes sense that RMS does not have this information: we use it to characterise the physical results of actions, and there is no corresponding notion in RMS. This has the effect of making the assumptive and realistic projections defined in chapter 4 exactly the same. If we assume that $e^{-} = \emptyset$, then there is no difference between assumptive and realistic e-element application. The only difference to begin with is that assumptive e-element application uses $e^{\pm}$ to generate follower cases, whereas realistic application uses only e^{+}. If all e-elements have an empty external-results postset, then the two methods of application are the same. Because of this, in what follows, we usually just say "projection" instead of "assumptive projection", or "realistic projection". If confusion can arise, simply insert the word "assumptive".

Another bit of the plan net framework that can be dispensed with in this section is the @ function. Recall that this function is used at plan execution time to map instantiated e-elements into addresses of the agent's primitive routines which implement the actions specified by the e-elements. This realises the idea of event executancy suggested in chapter 2. Since an RMS need not discuss plans, never mind plan execution, event executancy is irrelevant.

In his earlier work, Doyle included what were called *conditional proof* justifications, and we do not address such justification structures here. They are more complex than simple reasons, and have not been well integrated with other ideas in the literature. More thought is required to include such justifications in the plan net framework.

In the plan net formalism, e-elements correspond to Doyle's simple reasons, and b-elements correspond to the nodes or potential beliefs that simple reasons justify. Because Doyle no longer makes a clear distinction between potential beliefs and simple reasons, we will be forced to adapt some of his terminology and definitions when showing the relationship with plan net ideas. This has one very important result: in RMS, simple reasons can introduce other simple reasons, while in plan nets, e-elements can *only* introduce b-elements. This happens as a result of the distinction between b-elements, intended to represent potential beliefs, and e-elements, intended to represent the actions by which beliefs can be changed. In practise, the restriction we suggest does not seem to be a real limitation. The distinction is important to remember however, since in much of what follows, we will assume that simple reasons introduce only basic "nodes" (b-elements, potential beliefs), and that they never introduce other simple reasons.

Doyle says that an *assumption* is any node justified by a simple reason which contains a non-empty *out*list; that is, r is an assumption if and only if $B(r) \neq \emptyset$. To quote Doyle (1979, p.235): "An assumption is a current belief one of whose valid reasons depends on a non-current belief, that is, has a non-empty second set of antecedent beliefs.". Our notion of assumption (chapter 4, definition 20) is quite different from Doyle's. According to our definition, a *belief* is a b-element which is *in*, and which can be discharged by an e-element. A b-element can be discharged by an e-element if the e-element has concession in a case which does not contain the b-element, and which under realistic occurrence, can produce the b-element. If a b-element is *in*, but cannot be so discharged, then it is an *assumption*. This is different to Doyle's notion of assumption: we permit assumptions to be b-elements which are *in*, but have no reason (e-element) at all. Doyle's definition labels all *in* formulae justified by reasons with non-empty outlists as assumptions; we do not. According to our definition, it only matters whether or not the reason given is one which can account for the formula; the reason's *immediate* foundations are not considered. Doyle's assumptions are defined purely in terms of their underlying reasons. We allow for b-elements to be *in* independent of any strict reason: one way that b-elements can be made *in* is through an assumptive projection. Such a projection makes the formulas describing the external results of action *in*. According to our definitions (and Doyle's) these formulas have no

reasons to be *in* at all.

de Kleer (1984) equates assumptions with choices. It appears difficult to compare our notion of assumption with his, since what constitutes a choice will be determined by the overall problem solving architecture. In planning problems, choices arise in goal orderings, operator schema selections, and variable bindings. It would appear that de Kleer's definition of assumption subsumes ours, since it covers *all* choices, while our definition covers only certain limited situations.

Doyle (1979) defines a *premise* justification as a justification which has no nodes in either its *in*list or *out*list. That is, given a simple reason r, r is a premise justification if and only if $A(r) = \emptyset$ and $B(r) = \emptyset$. This concept of *premise* is analogously represented by an e-element e, where ${}^{\pm}e = \emptyset$. Such an e-element always has concession, and can make any b-elements in e^{+} *in* at any time.

The RMS ideas of *in* and *out* seem to be essentially those captured in definition 20 of chapter 4. However, unlike Doyle, we reserve the term "believed" for a b-element which is *in*, and in addition has a *reason* to be in.

Another basic difference between RMS and plan nets is that using plan net ideas, it is possible to give a *reason* for something being *out*. Such a reason comes in the form of a reconciliation set that would require removal were the formula in question to be added. It is not possible to give a reason for an assertion being *out* in RMS, since there is a fundamental asymmetry between *in*ness and *out*ness. In RMS, it is possible only to give reasons for making nodes *in*. In order to make nodes *out*, the application program utilising the RMS's services must introduce a special node, called a *contradiction*. RMS will take appropriate steps to ensure that this contradiction node is made *out*. The process by which this is done is known as *dependency directed backtracking*.

RMS is used by a problem solving program. Under RMS philosophy, it is the responsibility of the problem solver to signal any given node as a contradiction. It is possible to view the domain constraints we defined in chapter 4 as the *reasons* the problem solver would have for signalling

nodes as contradictions. Including these reasons in the basic definitions allows us to be explicit about whether any given formula is *not in* because the agent is "unaware", or *not in* for a good reason; that is, because the agent believes something to the contrary.

And those are the basic correspondences between Doyle's RMS and plan net ideas. In the next few sub-sections, we will examine in more detail how the functions supported by the two representations relate.

6.4.3. Truth maintenance and net projection.

The truth maintenance process makes any necessary revisions in the current set of beliefs when the calling program adds to or subtracts from the current set of justifications. Doyle (1979) presents an abstract algorithm which describes the process, and Charniak *et al* (1979) give a simple LISP implementation.

If we view a plan net $P = (B,E;F,A)$ as a collection of simple reasons, then we can imagine carrying out the truth maintenance process on it as follows. Let $X \subseteq \Pi(L_B)$ be a set of domain constraints. Picture an RMS based on the plan net formalism. Primitives would be available for modifying the net's e-elements: any calling program would be permitted to add e-elements (justifications) to the net, and to remove e-elements from the net. The calling program would also be responsible for specifying a current set of *assumptions*, as defined in chapter 4, and discussed above. Such a set of assumptions is simply a set of L_B formulae, given without justification. This set is a *case*, and we will refer to it in what follows as c.

In order to carry out the truth maintenance process, we need only construct a projection S_{PX} of the net P from the case c, respecting X. The nodes of the projection contain cases that are reachable under e-element occurrence, starting from c. Call a case *terminal* when in that case no net e-element has concession. Then all terminal cases in the projection are those in which the plan (*qua* net of justifications) is "stable". This means that for each of these cases, no more b-elements can be made *in* than are already *in* the case. This is because b-elements are only made *in* through the occurrence of an e-element, and in a terminal case, no more e-elements have

concession. The set of terminal cases in the projection therefore describes all legal assignments of *in*ness and *out*ness open to the truth maintenance process.

That's all there is to performing truth maintenance using a plan net. Of course, since only small parts of the net will be modified by the calling program at one time, it would make sense to perform some kind of sensitivity analysis, so that an entire projection need not be constructed. However in principle, following the addition or deletion of an e-element (justification), all one need do is construct a projection from the current set of assumptions using the current plan net in order to find all legal and stable *in* and *out* assignments.

As pointed out by de Kleer (1984, p.81), a standard RMS contains no useful notion of global state. All that RMS guarantees is that each justification is satisfied in some way. The reason that a projection is useful is precisely because it does contain a global notion of state. It contains a specification of *all* possible states. This means that doing truth maintenance via a net's projection actually overcomes all the problems related to the RMS "single state" problem, mentioned by de Kleer (1984, p.81; see also the 1986 papers for an extended discussion). de Kleer comments on the following difficulties with RMS.

(1) *The single state problem.* It is impossible to compare two equally possible solutions.

(2) *Overzealous contradiction avoidance.* Given a contradiction between two choices, RMS avoids including both. As de Kleer shows, it is important to be able to draw inferences from two contradictory choices independently.

(3) *Switching states is difficult.* There is no way to change a choice that is not forced on the problem solver by a contradiction. And even if one could change state to consider a different choice, there is no way to specify the target state.

(4) *The dominance of justifications.* de Kleer equates assumptions with choices. Any given assumption set represents some choices made by the reasoning component. This set can be freely changed. In RMS, there are only justifications.

These problems are overcome by having all satisfactory states of the RMS system available simultaneously. We can do this when using a plan net to represent a net of justifications, since all

possible net states are contained within the net's projection. We cannot address all the problems originally itemised by de Kleer, but we can deal with those given above. The solutions are as follows.

(1) *The single state problem.* If the appropriate interface primitives are supplied, the reasoning component can examine each of the net's terminal cases, and compare them easily.

(2) *Overzealous contradiction avoidance.* Under net projection, a possible contradiction (a domain constraint violation) is resolved by retracting those b-elements which can directly restore consistency, and creating a new state to represent the avoided contradiction. This allows the pursual of multiple, contradictory lines of reasoning.

(3) *Switching states is difficult.* In order to switch states (cases), one need only supply the appropriate interface for the reasoner. There is nothing conceptually difficult involved. All required information exists in the projection, and can be made available as necessary.

(4) *The dominance of justifications.* The plan net formalism is *not* dominated by justifications. A projection depends on three things: a plan (collection of justifications), an initial case (set of assumptions), and a set of domain constraints.

We said above that there are two main processes supported by RMS: truth maintenance, and dependency directed backtracking. How is this second process manifested in the plan net framework? In RMS, the reasoning component is held responsible for marking any given node as a *contradiction.* A contradiction is intended to indicate that the node's support contains contradictory beliefs; in order to correct the contradiction, RMS traces down the underlying assumptions (in Doyle's sense of the word) to find and remove at least one of the assumptions in order to make the contradiction node *out.* The details of this process as implemented in RMS are given in Doyle (1979, pp.18-21).

Contradictions are signalled in the plan net framework by the belief consistency maintenance function, μ. Recall how this function is expected to operate. It is defined with respect to a set of domain constraints, and is passed two sets of L_B formulae. μ is to return a set of possible reconciliations. Each reconciliation is a set, which if removed from the first of μ's arguments, will

make the union of both its arguments *consistent* with respect to the given domain constraints.

When creating a plan's projection, μ (in tandem with the reconciliation set selector, ψ) is used to define the appropriate next-state generator. Given a set of formulae in which a specific e-element has concession, the additions to possible successor states are found from the e-element's appropriate postset. No deletions are explicitly called for. Formulae are only deleted to obtain a possible successor through μ. The function is passed the enabling set of L_B formulae and the e-element's appropriate postset. μ will call for the deletion of only those formulae that are necessary to restore consistency to the enabling set of formulae, given the e-element's additions.

We can view this process as a degenerate form of dependency directed backtracking. Inconsistent cases are being avoided, much as would one would require of a dependency directed backtracking procedure. However, the way we reconcile inconsistencies is overly simplistic. The reconciliation set selected by ψ is *directly* removed to generate a successor case. Unfortunately, some formulae in the reconciliation set will likely be *in* the enabling case as the result of some chain of inferential reasoning. μ does not take this reasoning into account when calling for the removal of a formula. It would make more sense to require that μ trace down the *source* of the inconsistency. By tracing the inconsistency to its source, it would become possible to ensure that the reasoning by which it was caused cannot recur.

The idea of tracing down inconsistencies in a plan net is rather interesting. It seems that what is required is an idea of "inverse plan projection", or plan *retrojection*. Given a set of formulae to remove, the idea would be to retroject the net, in order to make the formulae *out*. The retrojection would bottom out when the net locks in a terminal (read initial) case. The cases derived in this way would become possible successors to the case which gave rise to the inconsistency. It is possible to view the case generator defined in chapter 4 as a degenerate version of this: the set subtraction operation in the definition of the next-state generator simply removes inconsistencies; the scheme just suggested would "undo" these inconsistencies, by unapplying the e-elements which gave rise to them. Of course, this is only a suggestion, and more work is required to explore the real issues involved in constructing plan retrojections to reconcile

inconsistencies.

If what we have defined is a special case of dependency directed backtracking, it seems reasonable to ask why it works at all. The answer appears to be that in planning, most of the e-elements contained within plans are there because of their external results. That is, most planned actions are intended to achieve their results in the environment, not in the agent. This means that few inferential actions occur in our plan nets: most *in* formulae are therefore *assumptions*, not reasoned *beliefs*. Because of this, in most plans, it is fine to simply remove inconsistencies directly. Since the formulae will probably not have arisen from any inferential activity, they can be directly removed. This direct removal is exactly what is done by the mechanism defined in chapter 4. As long as no chain of inferential reasoning lead to the inconsistency, direct removal of reconciliation sets appears to be a fair option. In general however, we would want the inconsistency avoidance mechanism to be more robust, and incorporate a bigger part of Doyle's ideas on dependency directed backtracking.

Node	Justification	Justification Name
1	(SL (3) ( ))	J1
2	(SL ( ) (1))	J2
3	(SL (1) ( ))	J3
4	(SL (2) ( ))	J4a
4	(SL (3) ( ))	J4b
5	(SL ( ) ( ))	J5
6	(SL (3 5) ( ))	J6

Figure 6.3: An example of Doyle's dependency relationships.

6.4.4. Representation: an example.

We now present a simple example of the basic correspondence between Doyle's (1982, 1983, 1985) RMS work and plan nets. In figure 6.3, we recreate the example given by Doyle (1979, p.241), and in figure 6.4, the corresponding plan net version of the same dependency structures is presented. For purposes of demonstrating the correspondence, we assume that no domain constraints are given, and that the initial set of assumptions (in the sense we have defined) is the empty set. Such a simplification is necessary, since RMS does not include the required concepts.

The interested reader can examine Doyle's original presentation for all the basic RMS dependency terminology. The only information really required to demonstrate the correspondence between RMS and the plan net formalism is the *support-status* of the given nodes. Using the dependency information of figure 6.3, RMS assigns nodes 2, 4, & 5 a support-status of *in*, and all other nodes are assigned a support-status of *out*.

For the corresponding plan net, we can use a projection to determine which b-elements (nodes) can be *in* and *out*, as follows. The initial case of the projection is the empty set, ∅. No domain constraints are required. This is fair, since the example as presented by Doyle is not intended to demonstrate the notion of dependency directed backtracking in response to a signalled contradiction. This means that no e-element occurrence can force *out* any b-elements. A graphical version of the plan's projection is given in figure 6.5. The two e-elements enabled in the initial case are *J2* and *J5*. The enable-out preset of *J2* is satisfied by ∅, since the b-element *1* is not *in* the case. *J5* has no preset, and so is also enabled. Under occurrence, *J2* adds the b-element *2*, and *J5* adds the b-element *5*, to the initial case S_0, to create the only successor, S_1. Thus the case depicted in S_1 shows the b-elements *2* and *5* as being *in*. In S_1, only *J4a* has concession. Under e-element occurrence, it adds the b-element *4* to S_1 to generate S_2. In this terminal case, only b-elements *2*, *5* and *4* are *in*, all others in the net are *out*.

6.4.5. Some corresponding definitions.

We can actually express the relationship between RMS and the plan net representation more precisely. In his later work, Doyle (1982, 1983, 1985) formalised much of what is required of a

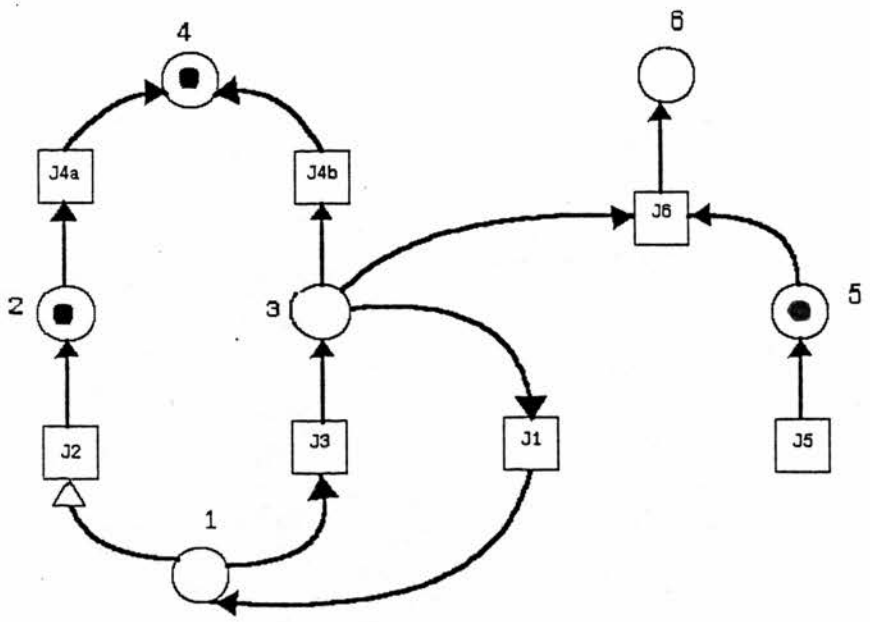


Figure 6.4: A plan net expressing the same dependency information.

general RMS. Using some of these formal ideas and the definitions of chapter 4, it is possible to show more rigorously some relationships between the RMS truth maintenance procedure and net projection. In particular, it is possible to show that what Doyle calls *strict groundedness* corresponds exactly to what can be called *reachability* in a net's projection. In this section, we present a proof of the correspondence.

Doyle (1982) presents a formal analysis of many aspects of simple reasons. In Doyle (1983), the major of these results are distilled down into a short paper, more accessible than the full original 1982 report. In Doyle (1985), many of the results of the 1983 paper are used to

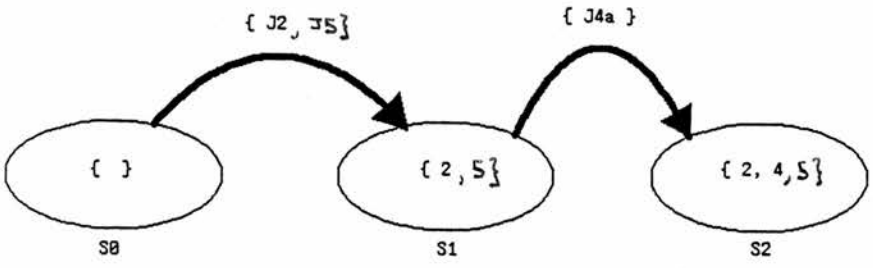


Figure 6.5: The projection of figure 6.4's plan from the case { }.

demonstrate how simple reasons can be viewed as economic, rather than logical entities. We adopt the notation and terminology of Doyle (1985), since it presents a compact and simple exposition of the underlying ideas.

Doyle (1985) suggests that the precise interpretation of a simple reason has two principal parts:

> The first partial interpretation is as one of the agent's self-specifications, as a possibly non-monotonic "closure condition" on the agent's set of beliefs. ... The second partial interpretation of a simple reason is as one of the agent's restrictions on derivability or arguability of conclusions, that is, as non-monotonic "inference records.".
>
> (Doyle, 1985, pp. 87-88)

The first partial interpretation is formalised by saying that a mental state is *admissible* (with respect to simple reasons) iff it satisfies each of its component simple reasons. A state S is admissible just in case for each simple reason $r \in S$, if $A(r) \subseteq S$ and $B(r) \cap S = \emptyset$, *then* $C(r) \subseteq S$.

There is an obvious corresponding definition for any given plan net. We can say that a case $c \subseteq L_B$ is *admissible* (with respect to a net P) iff there does not exist an e-element $e \in E_P$ such that e has concession in c. The idea is precisely the same as that captured in Doyle's definition. A mental state (set of "beliefs", or case) is admissible with respect to some simple reasons if and

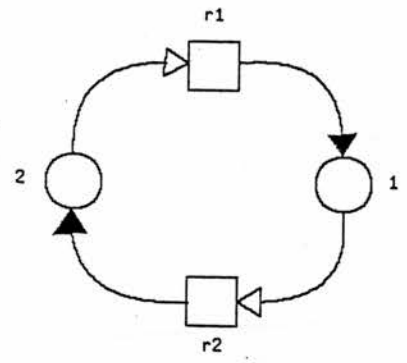


Figure 6.6: A plan net for Doyle's first example.

only if no simple reason can be "invoked" to add more beliefs to the state.

Doyle gives the following example to illustrate the idea of admissibility. Let $S_0 = \{\varnothing \,||\, \{1\} \,|\!\vdash\, \{2\},\ \varnothing \,||\, \{2\} \,|\!\vdash\, \{1\}\}$, where *1* and *2* stand for possible beliefs. Then S_0 is not admissible, but its three supersets $S_1 = S_0 \cup \{1\}$, $S_2 = S_0 \cup \{2\}$, and $S_{12} = S_0 \cup \{1,2\}$ are admissible.

A plan net expressing the simple reasons in Doyle's example is $P = (B,E;F,A)$, where

$B = \{1,2\}$

$E = \{r1,r2\}$

$F = (In0,In1,Out0,Out1)$

$In0 = \{ (2, r1), (1, r2) \}$

$In1 = \varnothing$

$Out0 = \varnothing$

$Out1 = \{ (r1, 1), (r2, 2) \}$

$A = \varnothing$

A graphical version of this net is given in figure 6.6. It is necessary to give the simple reasons "names", to allow the definition of the various ordering relations to go through.

As can be easily seen, in the case $S_0 = \emptyset$, both *r1* and *r2* have concession; thus S_0 is not admissible. In $S_1 = \{1\}$, and $S_2 = \{2\}$, neither *r1* nor *r2* have concession; thus, both these cases *are* admissible.

Doyle defines the second partial interpretation, that of simple reasons as restrictions on derivability of conclusions, in terms of a notion called "groundedness". Doyle explains this as follows. We say that a state S_2 is an *admissible expansion* of a state S_1 iff each element of S_2 is "grounded" in S_1. The two principle notions of groundedness are *local* and *strict*. In local groundedness, each element of S_2 has an immediate argument from S_1 and S_2, and in strict groundedness, each element of S_2 has a noncircular argument from S_1 alone.

Local groundedness can be defined as follows. S_2 is locally grounded in S_1 just in case for each $x \in S_2$, either $x \in S_1$ or $x \in C(r)$ for some $r \in S_2$ with $A(r) \subseteq S_2$ & $B(r) \cap S_2 = \emptyset$.

Doyle (1985, p. 87) gives the following example. Let $S_0 = \{\emptyset \,||\, \{1\} \,|\!\vdash\, \{2\}, \emptyset \,||\, \{2\} \,|\!\vdash\, \{1\}\}$, as before. If $S_1 = S_0 \cup \{1\}$, $S_2 = S_0 \cup \{2\}$, and $S_{12} = S_0 \cup \{1,2\}$, then S_1 and S_2 are each locally grounded in S_0, while S_{12} is not locally grounded in S_0.

We have not got a corresponding definition for plan nets; but one is easily constructed, by examining Doyle's. A few restrictions are first in order. Since we distinguish between e-elements (simple reasons) and b-elements (possible beliefs), we must define local groundedness for sets of b-elements (cases) in terms of collections of e-elements (plan nets). Thus given a net $P = (B,E;F,A)$, we can say that a case S_2 is *locally grounded* in S_1 using P iff $\forall x \in S_2$, either $x \in S_1$ or $\exists r \in E_P$: $x \in r^+$ & ${}^+r \subseteq S_2$ & ${}^-r \cap S_2 = \emptyset$. Given Doyle's examples, the corresponding constructions required for the plan net of figure 6.6 are obvious.

Much more interesting is the definition of *strict* groundedness. Using the theory of simple reasons, one can say that S_2 is *strictly grounded* in S_1 just in case for each $x \in S_2$ there exists a finite sequence $\langle g_0, \cdots, g_n \rangle$ of elements of S_2 such that $x = g_n$ and for each $i \leq n$, either $g_i \in S_1$

or there is some $j < i$ such that (1) $g_i \in C(g_j)$, (2) for each $y \in A(g_j)$, $y = g_k$ for some $k < j$, and (3) $B(g_j) \cap S_2 = \emptyset$. (This definition is essentially that given in Doyle, 1985, p.88.)

It is easy to create the corresponding definition using plan net notation. However, the interesting thing is that we already have the required notion of strict groundedness. All we've got to do it present existing ideas in the right form. Doyle's notion of strict groundedness is exactly the same as *reachability* in a plan net's projection. Reachability refers to whether or not a specified case is "reachable" from a given initial case. In terms of a plan's projection, this becomes a question of the existence of an appropriate path. Given cases c, $c' \subseteq L_B$, a plan net $P = (B,E;F,A)$, and the projection $S_{P\emptyset}$ of P from c respecting $\emptyset$, we can say that c' is *reachable* from c using P iff there exists a path of finite length from c to c' in $S_{P\emptyset}$.

What basis is there for equating reachability in a net's projection with strict groundedness? Given certain assumptions, it is possible to prove that the two notions are the same. In particular, we must assume that given two of Doyle's "states", S_1 and S_2, such that S_2 is strictly grounded in S_1, S_2 does not contain any more *simple reasons* than S_1; that is, if r is a simple reason, and $r \in S_2$, then $r \in S_1$. This restriction is necessary, since simple reasons can introduce new simple reasons; when using plan nets, this is impossible, since e-elements can only add b-elements, not more e-elements. This can be seen from the basic definition of a simple reason: $A \,||\, B \,||\!\!-\, C$ -- the set C is left unspecified, and is thus allowed to contain other simple reasons. We cannot use plan nets to do this, and in any case, the underlying utility seems rather suspect.

So the basic restrictions are as follows. Let S_2' be strictly grounded in S_1'. Assume that S_2' does not contain any simple reasons that are not also in S_1'. Let $S_2 = \{s \in S_2' \mid s \textit{ is not a simple reason}\}$, and let $S_1 = \{s \in S_1' \mid s \textit{ is not a simple reason}\}$. Let $R = S_1' - S_1$ be the set of all simple reasons in S_1 (and S_2). Having separated out the basic beliefs from the simple reasons used to derive them, we can now say that S_2 is *strictly grounded in* S_1 *by* R. Assume that for each simple reason in R, there is a corresponding e-element in the net $P = (B,E;F,A)$, with the flow relation defined appropriately to reflect the basic justifications expressed by the simple reasons in R. Given these conditions, it is not hard to show that the

following is true: S_2 is strictly grounded in S_1 by R iff S_2 is reachable from S_1 using P. The proof is given below.[1]

(IF part.) Assume that S_2 is strictly grounded in S_1 by R. Then for each $s \in S_2$, either s is in S_1 or there exists a sequence of elements of $S_2 \cup R$, $\langle g_0, \cdots, g_n \rangle$, such that $s = g_n$. The existence of this sequence implies the existence of a path from S_1 to S_2 in the projection as follows. For all $i \leq n$, either $g_i \in S_1$ (trivial case), or g_i comes from a *simple reason* g_j in R (by condition 1). By assumption, for each simple reason, $r \in R$, we have a corresponding e-element $e \in E_P$. Thus there exists an e_j such that $g_i \in e_j^+$. By conditions 2 and 3, we know that this e-element will eventually have concession, since 1) $\forall y \in {}^+e_j$, $y = g_k$, $k < j$ implies that all of the e-element's enable-in preset is eventually achieved, as specified by the sequence; and 2) since ${}^-e_j \cap S_2 = \emptyset$; that is, since in S_2 none of the e-element's enable-out postset occurs, none of it ever occurred, since without domain constraints, b-elements are never removed. If any of ${}^-e_j$ were ever added, they would remain in S_2. Thus the existence of the sequence implies the existence of a

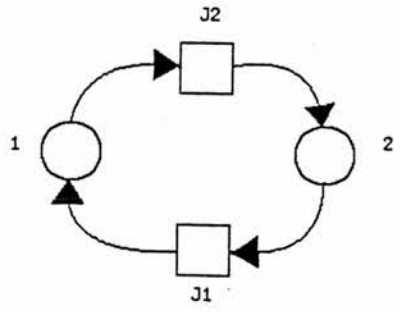


Figure 6.7: A plan net for Doyle's second example.

[1] The "conditions" the proof refers to are those given in Doyle's definition of strict groundedness, presented earlier.

path from S_1 to S_2 in $S_{P\emptyset}$, since each e-element required to produce an $s \in S_2$ is eventually enabled. This means that S_2 is reachable from S_1 using P.

(ONLY IF part.) Assume that S_2 is reachable from S_1 using P. Then there exists a path in the projection $h_1 l_1 \cdots h_n$ such that $S_1 = h_1$ & $S_2 = h_n$. This path can be used to construct sequences of elements of h_i and l_i $(1 \le i \le n)$. For each $s \in S_2$, either $s \in S_1$, or there will exist a sequence satisfying conditions 1, 2, and 3; namely, there will exist a sequence $<g_1, g_2, \cdots, g_n>$ where each $g_i \in \bigcup_{j=1}^{n} h_j \cup l_j$ such that $g_i \in S_1$ or there is an e-element g_j $(1 \le j \le n,\ j < i)$ such that $g_i \in g_j^{\dagger}$. Since g_j occurs in the sequence, either its preset is satisfied in S_1 or some intervening e-element satisfies it. In either case, we can guarantee that for all

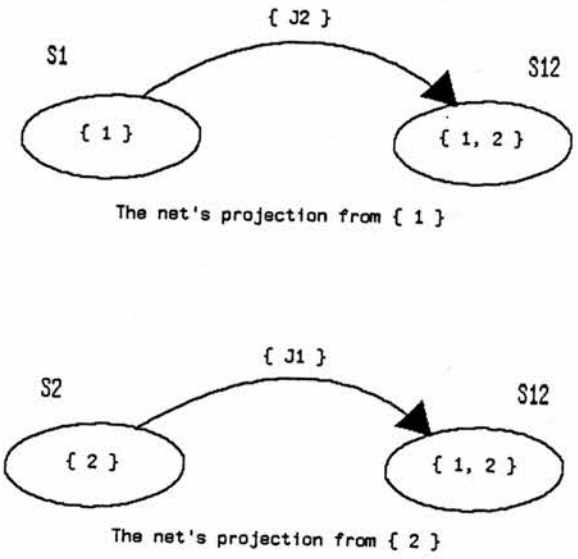


Figure 6.8: Two projections of the net in figure 6.7.

$y \in {}^{+}g_j$, $y = g_k$, $k < j$ & ${}^{-}g_j \cap S_2 = \varnothing$, or else g_j would not be in the path, since it could never have concession. Thus the existence of a path from S_1 to S_2 implies the existence of a sequence satisfying conditions 1, 2, and 3 in Doyle's definition of strict groundedness for every $s \in S_2$.

Examples are always useful where notation obfuscates. Doyle (1985, p.88) presents a simple reasons example, and we give it according to our slightly modified definitions, as follows. Let $S_0' = \{\{1\} \,||\, \varnothing \,|\!\vdash \{2\},\ \{2\} \,||\, \varnothing \,|\!\vdash \{1\}\}$. We must now separate out the simple reasons of S_0'. Let $S_0 = \varnothing$; i.e. let S_0 be the set of all elements of S_0' that are not simple reasons. Here, all elements are simple reasons, so nothing remains after this filtering operation. Let $R = \{\{1\} \,||\, \varnothing \,|\!\vdash \{2\},\ \{2\} \,||\, \varnothing \,|\!\vdash \{1\}\}$; i.e., let R be all the simple reasons in S_0'. Let $S_1 = S_0 \cup \{1\}$, $S_2 = S_0 \cup \{2\}$, and $S_{12} = S_0 \cup \{1, 2\}$. Then S_{12} is locally grounded in S_0 by R; S_{12} is not strictly grounded in S_0 by R; and, S_{12} is strictly grounded in both of S_1 and S_2 by R.

A plan net for this second example of Doyle's is given in figure 6.7. As indicated in the figure, there are two e-elements, representing the two simple reasons given by Doyle. We have called these *J2* and *J1*. *J1* has as its enable-in preset {2}, and *J2* has {1}. *J1* has as its internal-results postset {1}, and *J2* has {2}. Notice that in the case ∅, neither of the e-elements has concession. However, in the case {1}, *J2* has concession, and can make *2* in; in the case {2}, *J1* has concession, and can make *1* in. Two projections reflecting this are given in figure 6.8. By examining the two graphs, it is clear that the case {1, 2} is reachable from either {1} or {2}. Thus S_{12} is strictly grounded in both S_1 and S_2 using the given net. S_{12} is not strictly grounded in S_0 using the net, since no path exists in the projection of the net from ∅. However, as can be seen from the definition of local groundedness that we copied from Doyle, S_{12} is locally grounded in S_0 using the plan net given in figure 6.7.

6.4.6. Using multiple states.

As de Kleer (1984) points out, it is sometimes necessary to have available a number of possible solutions, so that these solutions can be compared. It is also occasionally nice to be able to find *all* possible solutions. Both of these are difficult or impossible with traditional RMS ideas, since there is no coherent notion of global "state". In this section, we present two brief examples to

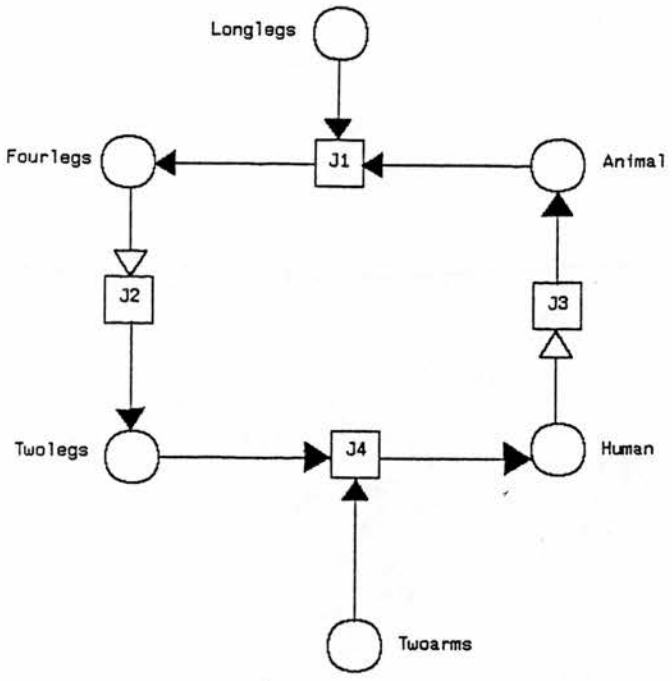


Figure 6.9: A net for classifying an object.

show how the plan net representation, when applied to the reason maintenance problem, allows the reasoning component to consider as many solutions as found necessary.

There will often be mutually contradictory methods available for solving a given problem. In certain circumstances, it might be desirable for the reasoner to consider all alternative solution strategies, even though they are incompatible. The net given in figure 6.9 depicts such a situation. The problem the net is designed to solve is one of classification. The system must classify a given object according to certain parameters. The parameters available are: whether the object has two

legs (*Twolegs*) or four legs (*Fourlegs*), and whether it is an animal (*Animal*) or a human being (*Human*). The symbols *Twolegs*, *Fourlegs*, *Animal*, and *Human* are predicates of zero arguments. The initial assumptions, as shown in the net, are that the object has long legs, and two arms.

There are four inferences modelled in the net. E-element *J1* describes the inference "if it is an animal, and has long legs, then it has four legs". *J4* describes the inference "if it has two legs, and two arms, then it is a human". Now while these might not be *sound* rules of inference (i.e., they do not represent deductions) they certainly represent plausible rules for deriving new beliefs from old. E-elements *J2* and *J3* describe default inferences. If it is not believed (or assumed) that the object has four legs, then it is all right to believe that it has two legs (by *J2*), and if it is not believed (or assumed) that the object is human, then it is all right to believe that it is an animal. The domain constraints given in the figure indicate that no object is both human and animal, and that no object has both two legs and four legs.

A projection for the net is given in figure 6.10. This projection shows all possible executions for the plan net. Starting in S_0, it is possible to apply *J2* and *J3* in any order, resulting in S_2. In S_2, the agent believes that the object has long legs, two arms, two legs, and believes that it is an animal. So far, beliefs have only accreted monotonically, since no domain constraints have been violated by formula additions. In S_2, two e-elements have concession: *J4* and *J1*. The occurrence of *J4* adds the belief that the object is human. Since the addition of the formula *Human* to the case in S_2 violates one of the domain constraints, possible successors to S_2 under *J4* are found by removing reconciliation sets suggested by μ. Here, there is only one alternative, and thus only one possible successor. Removing {*Animal*} will restore consistency, and results in the case shown in S_4. Similarly for the occurrence of *J1*: it causes a constraint violation, and the only way to reconcile the difficulty is by removing the set {*Twolegs*}. This indicates that any sequence of e-element application which runs through S_2 models a mixing of two incompatible inference procedures. The two successors to S_2 represent the states of belief that are possible ways of reconciling the incompatible results of the inferences. It would appear that the normal route through the projection is from S_0, through S_1 to S_4, or from S_0 through S_3 to S_5. Each of these paths in isolation seems to model some method of continuous reasoning, while any route through

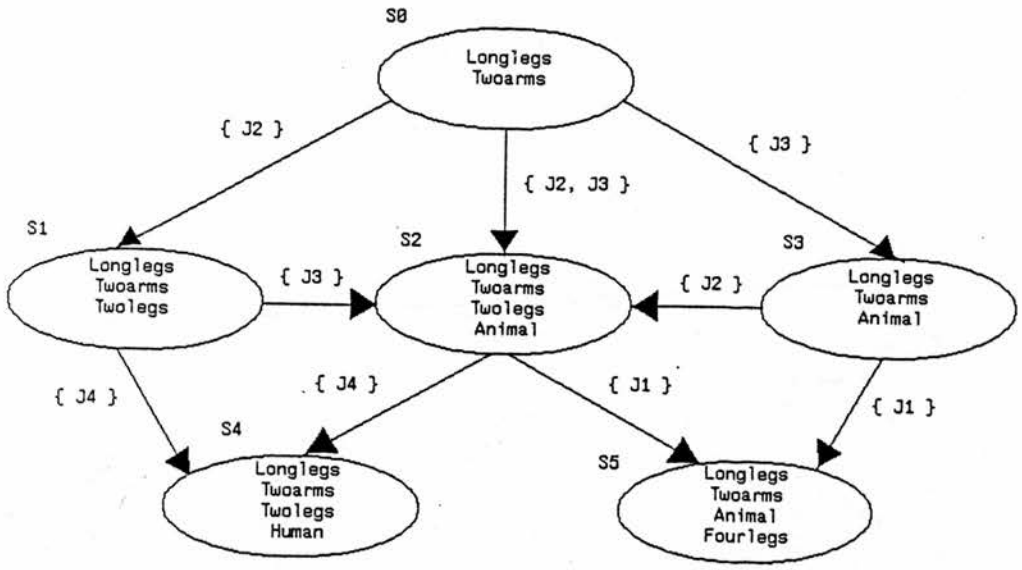


Figure 6.10: Possible object classifications.

S_2 is mixing the two methods together.

The important issue is that while the two methods of reasoning about the nature of the object are *incompatible*, it is actually possible to describe both in the same structure. One plan net can be used to represent the justifications required by both methods of reasoning, and the net's projection can show how the methods can be mixed and separated to produce consistent sets of beliefs. In the example, there are two final states for the given plan net: S_4 and S_5. Both are possible, if incompatible, states of belief following application of the actual rules of inference modelled by the net's e-elements.

Our second example comes from de Kleer (1984, p.82). He shows how his assumption-based method of reason maintenance cannot support legitimate conclusions based on assumption sets which have previously been used to derive contradictions. Suppose that a problem solver had deduced that

(1) $x = 1$, under assumption A;

(2) $x + y = 0$ under assumption B;

(3) $z = 1$ under assumptions A & B; and,

(4) $z = 0$ under assumptions A & B.

In de Kleer's system, $y = -1$ is not derivable from (1) and (2), as the assumption set $\{A, B\}$ is contradictory. However, if $x = 1$ is derived later, and does not use assumption A, then $y = -1$ *is* derivable. This problem is known as *unouting*. It also occurs in Doyle's RMS, in a slightly different form.

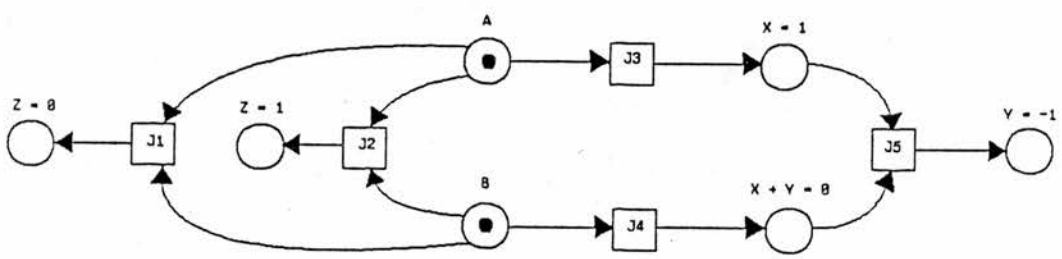


Figure 6.11: Doing simple mathematics.

We offer a plan net solution to this problem in figure 6.11. The e-elements in the plan net represent algebraic inferences, and the b-elements represent potential beliefs about algebraic quantities. The assumption set $\{A, B\}$ is shown graphically in the figure. Since it doesn't matter what these assumptions are, we leave their exact content unspecified, as does de Kleer. *J2* represents the derivation of $z = 1$ from the assumptions, and *J2* represents the (contradictory) derivation, $z = 0$, using the same assumptions. The e-elements *J3* and *J4* represent the derivations of $x = 1$ and $x + y = 0$, respectively. These results are used in the derivation of $y = -1$, modelled by *J5*.

For the purposes of plan net projection, we assume that the set of domain constraints is $\{\{V = A,\ V = B\}\}$, which can be taken to mean that each variable can only be believed to have one value at a time. How can we find out what states of belief are possible, given the net and initial set of assumptions? Once again, the plan's projection gives us all possible net states. For this plan, the projection is rather large, and contains 15 different cases. We show all of these cases in figure 6.12, but omit some of the steps between cases for clarity. There are no terminal cases in the projection of the plan from the assumptions respecting the given domain constraints. This is a result of the derivations modelled by *J1* and *J2*: they always have concession, and on firing, force out a belief regarding the value of the variable *z*. This can cause oscillation between the left-hand and right-hand sides of the projection. It is these left-to-right and right-to-left transitions that are not shown in the figure. A transition is *always* possible from one case in the right sub-graph to the corresponding case in the left sub-graph, and vice versa. Transitions from the central sub-graph are also always possible to either of the left or right sub-graphs. The left hand sub-graph of the projection represents those cases in which $z = 0$ is believed, the right hand sub-graph represents those cases in which $z = 1$ is believed, and the central sub-graph represents those cases in which there are no beliefs about the value of *z* at all.

The projection thus captures all possible states for the algebraic derivations modelled in the plan net of figure 6.11. This works because we are not saying that a set of *assumptions* is inconsistent (as does de Kleer), but rather, find inconsistencies and reconcile them only when necessary. Now in the example, this only succeeds because the inconsistent values for *z* are

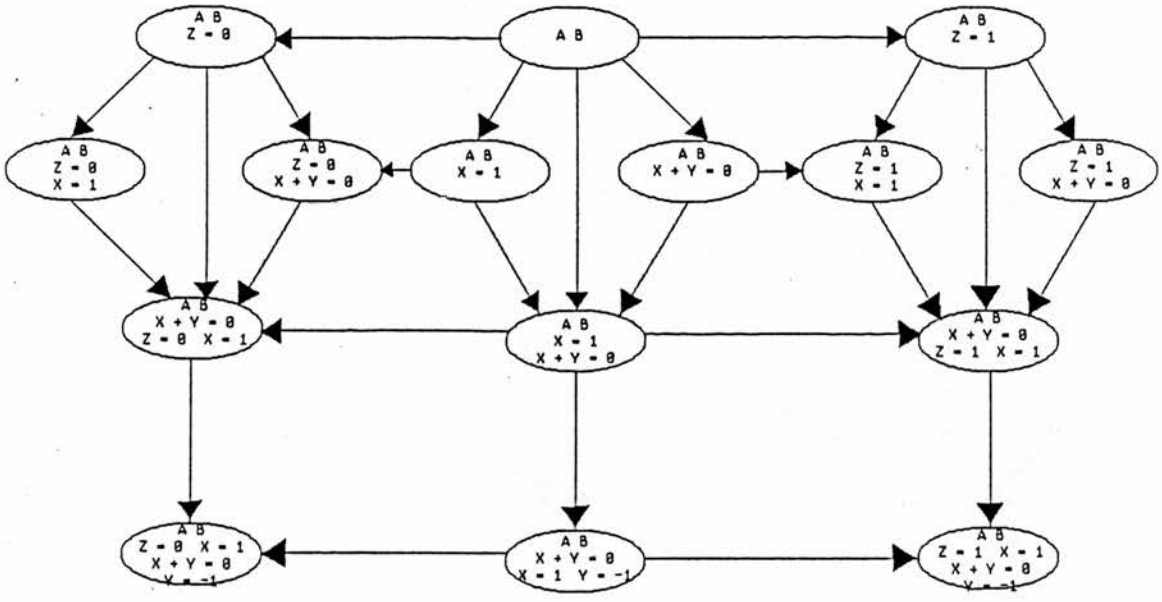


Figure 6.12: What the derivations can come up with.

derived in a single e-element application. Single step derivation means that simply removing inconsistencies is equivalent to tracing them down to their roots. For more robust inconsistency reconciliation, it is necessary to define a true dependency directed backtracking procedure, as we have discussed above. However, the example does serve to show the underlying utility of the ideas.

6.4.7. Some general comments.

It seems that plan nets can represent the notion of simple reasons first advanced by Doyle (1985), and also provide the multiple-state feature of de Kleer's (1984, 1986a, 1986b, 1986c) approach to reason maintenance. While the examples of this section should serve to demonstrate the underlying correspondences, more work is required to make plan nets truly useful for doing serious reason maintenance work. In particular, we need to consider whether or not separating potential beliefs (b-elements) and the means by which these beliefs may be acquired (e-elements and the idea of e-element occurrence) has any effect on what sort of systems we can represent. It seems that more experience is needed with real systems based on the plan net representation to answer this question.

Another issue that must be addressed is that of dependency directed backtracking. The plan net approach to reason maintenance will not be adequate until some form of dependency directed backtracking is integrated with the ideas presented in chapter 4 and informally extended in this chapter. All the examples we have presented depend for their success on the fact that detected inconsistencies do not arise from long inferential "chains". This seems fine for many planning situations, where there are more assumptions regarding the external outcome of actions than there are true justified beliefs. To use the plan net representation more generally, some precise formulation of dependency directed backtracking is necessary.

6.5. Teleological information (Goal Structure & Plan Rationale).

Sacerdoti's (1975a, 1975b) work on NOAH was influenced by Tate's (1974, 1975a, 1975b) research on *goal structure*. NOAH contained something called the Table of Multiple Effects, or *TOME*, which was based directly on Tate's (1974) original idea of holding periods. Tate's (1977) Nonlin planner built on NOAH's action-ordering plan representation, and extended it by integrating richer goal structure information. Nonlin used this information to guide the suggestion of possible action re-orderings following conflict detection. Recent systems, such as Deviser (Vere, 1981, 1985) and SIPE (Wilkins, 1983, 1984a, 1984b) have also started to include such information.

What is goal structure? Essentially, it is information expressing the "causal connectivity" or *teleology* within a plan. Goal structure determines which action results are necessary for the successful continuation of a plan, and which results are mere epiphenomena. Analysis of a plan's goal structure allows plan modification suggestions to be based on an understanding of these necessary causal connections and irrelevant effects. The information permits plan projection reasoning to determine how one action can be used to enable another.

Goal structure is exactly what plan nets are constructed from. The ordering relations used to link together b-elements and e-elements describe very precisely the relationships between actions, preconditions, and results. As noted by Tate (1984c), it is important to realise that *operators* also have an internal goal structure. This result falls naturally out of the plan net framework with no effort. In principle, it is not necessary to make any distinction between plans and operators. In chapter 4, we did make a distinction based on the number of e-elements a net contains: operators were allowed to contain at most one e-element, while plans could contain an arbitrary number. This distinction only arises from a desire for simplicity of operator analysis, and does not reflect any underlying restrictions on the definition of an operator. If one is willing to allow arbitrarily complicated nets to be operators, then operators will include all the required goal structure information.

Tate's goal structure does contain some information not included in the basic ordering relations of a plan net. Goal structure includes the idea of "typing" conditions, to give an indication to the planner of whether a condition can be viewed as a sub-goal, or should be considered merely as a "context" condition. As Tate (1976, 1977, 1984c) suggests, operator "preconditions" are often only included to draw information from the *context* of the operator's application. It would make no sense for the planner to attempt to *achieve* these sorts of conditions. The issue of such contextual preconditions is further addressed by Charniak and McDermott (1985). This idea of condition typing is *not* included in the plan net formalism. Such information, or some variant thereof, is required to make plan *construction* efficient. We return to this issue in chapter 8, where suggestions are made regarding options for future research.

Wilkins' SIPE planner (1983, 1984, 1985) also includes an account of a plan's internal teleology. In SIPE, this is called "plan rationale". The idea as employed in SIPE is much the same as used in Nonlin. As defined by Wilkins, plan rationale refers to (the main reason) "why" any given action is in a plan. (See Wilkins, 1984a, p 283.) Wilkins suggests that plan rationale is useful for determining condition holding periods, for finding out what changes in a plan could pose problems, and for determining the relationship between components of a plan at different hierarchical levels. The ordering relationships in a plan net explicitly address the first two of these uses; they cannot address the last, since there is no notion of action hierarchy in a plan net. If the framework were to be extended to include such a notion, it seems as through the existing ordering relations would do the job of Wilkins' plan rationale. Such a claim can only be borne out by the appropriate extension of the plan net definition.

6.6. A simple STRIPS-like construction algorithm.

This thesis focuses on *plan representation*. But much of the classic planning problem actually deals with plan construction; in particular, efficient plan construction performed in terms of some specified goals. This section addresses the automatic construction of plan nets, but does so without claiming any great improvement in efficiency over existing techniques. We consider only a relatively simple construction algorithm that has clear limitations. Recent work by Chapman (1985) deals with better plan construction algorithms, as does the excellent report by Pednault (1985). This chapter offers no novel plan construction ideas, except those that pertain to the construction of plans to *sense*. Some comments about the construction of iterative plans are made, but the reasoning required to build iterative plan net constructs remains largely a research task. We also do not address the construction of disjunctive plans, and we ignore the problem of planning for goals of avoidance. With respect to this last point: this means that the construction method we present completely ignores all e-element enable-out presets. In general, this is a rather serious omission, and future work on plan net construction should consider the problem of planning for inhibitory pre-conditions. No claims are made for the completeness of the algorithm; that is, no guarantees are given regarding termination. However if a solution *is* found it will be

correct, but we offer no proof for this.

Here are some more brief caveats regarding the algorithm. We ignore the issue of constructing projections which consider potential parallelism. This is an important issue, and requires further thought. Related to this, we leave open the issue of how to construct sound execution advice. Some alternatives for doing this are clear, but we do not present any. We also assume (for simplicity) that operators are nets with single e-elements. This actually follows on definition 8 of chapter 4, so is not truly an assumption. For current purposes, operators must be nets with only one e-element. This makes net construction much simpler.

STRIPS proceeds in a manner dictated by the GPS means-ends analysis (MEA) heuristic. Given an initial state and a set of goals, STRIPS operates by using MEA to find an operator relevant to reducing a difference between the initial state specification and goal set. The preconditions of the selected operator are used as a new goal set, and MEA proceeds recursively. The MEA process bottoms-out when an operator is found which is applicable in the problem's initial state specification. The selected operator is applied, and the resulting state is used to find remaining differences, on which the MEA heuristic is once again employed. STRIPS can be considered to be doing *forward* state-space search, through a space of possible *world* states: it begins with a specification of the initial world state, and applies operators until a state description satisfying the goal specification is produced. MEA can be thought of as simply a *heuristic* for determining which operator to apply to any given world state description.

Notice that by basing our algorithm on STRIPS, we inherit all the problems associated with over-eager goal achievement, such as the inability to optimally solve certain blocks world problems, and the inability to cope with tasks involving actions which are not obviously directed towards goal achievement (such as the register swapping problem). In terms of the discussion of chapter 3, by using MEA, we are making the linear assumption. However the algorithm *is* reasonable as a first approximation to a general plan construction method.

It is simple to adapt the basic STRIPS algorithm to generate plan nets. We must add to the algorithm the notion of *goal consistency*. STRIPS operates under the *closed world* assumption, in

that all features relevant to a problem's solution are assumed to be given in the initial state specification. Since we are interested in planning to sense, our construction algorithm cannot make this assumption. Given a goal, our plan construction reasoning must consider whether or not the goal is consistent with the initial set of beliefs, i.e., consistent with the initial state specification. Using the ideas developed at length in chapter 4, this can be phrased as the problem of checking whether the union of the singleton set containing a goal and the initial world model is consistent with respect to some declared domain constraints. Consider what such a consistency check means. If the goal *is* consistent with current beliefs, then the agent is simply ignorant of whether or not the existing state of the world currently satisfies the goal. The goal may be *true* or *false* with respect to the agent's current environment, but no beliefs are held by the agent to indicate this. Thus, if a goal *is* consistent with the agent's existing beliefs (with respect to some domain constraints), then what is required is a plan which can cause the agent to come to believe the required goal. If the goal *is not* consistent with the agent's existing beliefs (with respect to some domain constraints), then a plan must be constructed to change the state of the world so as to bring the required goals about.

The basic STRIPS algorithm modified to take goal consistency into account is given in figures 6.13, 6.14, and 6.15. The basic architecture of the algorithm is as follows. There are a number of global data structures available. These include the plan net, $N = (B,E;F,A)$; the plan net's

---

```
planner( goals )
  begin
     build-net( goals );
     fix-execution-advice( goals );
  end.
```

Figure 6.13: A simple method for building plan nets.

---

assumptive projection, $S = (H,R)$; a set of domain constraints, X; and a set of operators O. Initially, the plan net is empty (each component of the plan net tuple is empty); the projection contains only the problem's initial case; the domain constraints are those required by the problem; and the set of operators are those that can be used in deriving a potential solution to the problem.

In the algorithm and ensuing discussion, we often fail to distinguish between the program variable used to refer to any particular node of the projection graph, and the actual case contained in the node. In an implementation, such a confusion could not be tolerated.

We assume that each case in the globally available assumptive projection has a status associated with it; either *open* or *closed*, used to indicate whether or not the case is worth future consideration. If a node is *open*, it is deemed worthy of consideration, if *closed*, it is not. A node has its status set to *open* when initially created, and set to *closed* by an evaluation heuristic. Once closed, a node is never re-opened. This destroys the completeness of our search, but is necessary for coping with complexity.

The following routines are assumed to be available and to have the described functionality.

(1) *fix-execution-advice(goals)*. Accepts a set of L_B formulae, *goals*, as its single argument, and modifies the execution advice (the *before* order) of the plan so as to make the plan's advice *sound* with respect to *goals*.

(2) *no-unexplored-open-cases-left( )*. Is a predicate of no arguments, and returns *true* if there are no cases in the projection from which any of the provided operator schemas would have concession if appropriately instantiated which do not already exist in the plan net. Thus, *false* is returned providing that 1) there is an operator containing an e-element, e, for which 2) a projection case c, and binding β can be found, such that 3) $\hat{\beta}(e)$ is *c–enabled*, and 4) $\hat{\beta}(e)$ is not already in the plan net. Otherwise, *true* is returned, indicating that none of the operators, in none of the open projection cases, under no substitution of constants for variables has concession, such that the operator has not yet been tried (does not exist in the plan net).

(3) *terminate-with-failure( )*. Accepts no arguments, and is used to halt plan construction, indicating failure.

(4) *choose-open-case( )*. Returns a case specifier which refers to a case in the plan net's assumptive projection. The designated case will be one which is unexplored, and open, in the sense described above. That is, there will exist an operator, and a binding, such that in the case designated, and under the binding, the operator will have concession. The operator is not determined by *choose-open-case*; this routine simply returns some case from which progress is considered possible. Thus, this routine embodies a heuristic about which case to proceed from, given a choice. A variety of heuristics can be imagined, ranging from simple breadth-first to A^*.

(5) *choose-goal(case1, case2)*. Accepts two sets of L_B formulae: the first is a case from the projection, and the second is a set of goals. It returns a single L_B formula which is a goal with respect to *case1*. This could be implemented as a simple set membership check: a goal is a formula in *case2* that is not also in *case1*.

(6) *consistent(case1, case2, constraints)*. Accepts two sets of L_B formulae and a set of domain constraints (a subset of the powerset of L_B). It returns *true* iff the union of the two sets is consistent with respect to the domain constraints, otherwise, it returns *false*.

(7) *MEA-choose-operator(goal, case, purpose)*. Accepts three arguments: a goal (a single L_B formula), a case (a set of L_B formulae), and a flag indicating the purpose of the operator. The flag can be set to one of *internal* or *external*. The constant *external* indicates that an operator relevant to causing the required goal be used to initiate means-ends analysis, and *internal* indicates that an operator relevant to causing a belief about the goal be used. In the first case, an operator will be selected on the basis of its declared external-results postset, and in the second, it will be selected for its declared internal-results postset. Notice that in the tradition of means-ends analysis, the operator directly relevant to reducing the difference is not returned, but instead, an operator which is expected to be of some use in getting to a position in which the operator of relevance can be applied, and the difference thus reduced.

The routine is structured to use the preconditions of the operator chosen for its appropriate postset as "goals" to backward chain on, eventually terminating with an operator which is applicable in the argument *case*. This operator has all variables substituted with constants as appropriate, determined by constants propagated back through the means-ends analysis, or by constants found in the given case. For more on how this is expected to work, see Raphael (1976), or Nilsson (1980).

(8) *randomly-choose-operator(case)*. This routine accepts a case (a set of L_B formulae) as its single argument, and returns an operator which has concession in the case. The operator is derived from the globally available operator schemas by substituting constants for an appropriate operator schema's variables. The operator returned is guaranteed to have

---

```
build-net( goals )
 begin
    until ∃ β ∃ h ∈ H: β̂(goals) ⊆ h do
    begin
       if no-unexplored-open-cases-left( ) then terminate-with-failure( );

       c := choose-open-case( );

       g := choose-goal( c, goals );

       if consistent( {g}, c, X ) then op := MEA-choose-operator( g, c, internal )
       else op := MEA-choose-operator( g, c, external );

       if op = ∅ then op := randomly-choose-operator( c );

       add-operator-to-net( op );

       do-assumptive-projection( c )
    end
 end.
```

Figure 6.14: The core of the simple plan net construction algorithm.

---

concession in the case given as an argument.

(9) *add-operator-to-net(operator)*. The single argument is an operator, and is inserted into the developing plan. Insertion of an operator is accomplished by adding each of the operator's components to the appropriate components of the plan.

(10) *do-assumptive-projection(case)*. Accepts a case as its single argument and uses it to extend the current assumptive projection. Only cases which have their status set to *open* are considered; all others are left unexplored.

(11) *worth-pursuing(case)*. Accepts a single argument which is a case, and returns *true* iff the case is deemed worthy of further consideration; otherwise, it returns *false*. This predicate embodies the heuristic used to terminate unlikely avenues of exploration.

(12) *e-elements-with-concession(case)*. Accepts a single argument which is a case, and returns the set of e-elements in the plan which have concession in the case.

(13) *choose-successor(set-of-cases)*. Accepts a set of cases (a subset of the powerset of L_B), and returns a single case. This function encodes the heuristic about which state of belief to adopt, following on a contradiction. As explained in chapter 4, the idea is that given any particular inconsistency, there will in general be many ways to resolve it. This function chooses a single successor from among those that are possible.

(14) *assumptive-occurrence(operator, case)*. Accepts an operator and a case, and returns a set of cases (a subset of the powerset of L_B). Each of the cases in the set returned is a possible successor to the case argument under the assumptive occurrence of the operator given.

(15) *add-arc-to-projection(case1, case2)*. Accepts two cases as arguments, and adds a link in the assumptive projection being constructed from the first to the second. If the second case does not yet exist in the projection, a graph node is created for it, and the status of the node set to *open*, indicating that further exploration is merited (at least until the state is closed when given the thumbs-down by the function *worth-pursing*).

(16) *status(case)*. Accepts a case, and returns its current status, either *open* or *closed*.

(17) *set-status-to-closed(case)*. Accepts a single case argument, and sets its status to *closed*.

With these primitive routines in hand, we can now describe the operation of the basic plan net construction algorithm. The top-level routine of figure 6.13 simply constructs a plan net, and then makes the net's execution advice sound with respect to the given goals. The real plan net construction work is done by the routine *build-net* of figure 6.14. This routine is iterative in its structure: it will continue work until a case exists in the plan's assumptive projection that satisfies the goal specification. Satisfaction means finding a case which, under an appropriate binding, includes the set of goals as a subset. Until this condition obtains, plan construction continues.

The main loop of the *build-net* routine operates as follows. If there is no way to proceed, then the algorithm exits without finding a solution. (Notice however, that the calling routine takes no account of such a possibility.) Provided that *open* options exist, work continues. An open case is chosen (perhaps by a best-first heuristic, or A^*), and used to select a goal from the goal set. Notice that since a formula is only a goal with respect to some "current" case, the routine *choose-goal* must be passed the case, *c*, and the set of goals.

The next step illustrates the use of some of the plan net definitions of chapter 4 to determine the sort of result required to obtain the goal. If the selected goal is consistent with what is currently believed by the agent, then the agent is simply ignorant of whether or not the goal currently obtains in the environment. If so, an operator is required which can cause a belief about the required environmental condition. This operator selection is carried out by the routine *MEA-choose-operator*. It must be passed the goal, the selected case, and a flag indicating the purpose of the operator. If the flag is *internal*, an operator is chosen on the basis of its internal-results postset; if the flag is *external*, an operator is chosen on the basis of its external-results postset. Notice (again) that an operator which is *directly relevant* to achieving the goal is not returned; rather, an operator which is expected to be *en route* to the enablement of such a directly relevant operator is found.

The next statement is a catch-all. Using MEA, we might fail to find an operator relevant to reducing the given difference. However, an operator is guaranteed to be applicable, since the predicate *no-unexplored-open-cases-left* was false on loop entry. Thus some operator is applicable to the case *c*, and MEA cannot find it. If this happens, a random selection of applicable operators is made. Notice that following operator selection, the program variable *op* is assumed to be set to the operator derived from a schema by an appropriate substitution of constants for variables. Thus, *op* represents an individual operator, *not* an operator schema. Following selection, the instantiated operator is inserted into the developing plan net. While we are not being very precise about the nature of operator addition, what is required seems intuitive. The operator's components will be unioned in with the existing plan net components, and the resulting structure will become the new plan net.

Following operator insertion, an assumptive net projection is carried out. This operation forms an integral part of the plan net's use in avoiding redundant work, since the plan net stores the results of previous operator applications. If any work was carried out previously, and is recorded in the plan net, then provided that the operators' preconditions are satisfied by the chosen case, their results will be immediately available, without further work. Existing work is used in a new context, thus furthering efficient plan construction. Also notice that the plan net is never cut down in size: operators are only ever added. This means that old work is never thrown away, and if it ever becomes relevant, is immediately available by virtue of the definition of case concession.

The routine *do-assumptive-projection* uses the globally available plan net and domain constraints to add on to the current assumptive projection. The projection is extended from the given case only if it is deemed worthy of exploration. If not, the routine sets the node's status to *closed*, and exits. If the node is worth pursuing, then the projection graph is extended in the obvious way, by finding those e-elements in the plan which have concession in the given case; for each e-element that does have concession, an assumptive successor is chosen, and added to the projection. If the successor case already exists in the projection, the routine *add-arc-to-projection* only adds an arc in the projection from c to c'. If, however, c' is not yet in the projection, then a new node structure is created for c', and this node's status is set to *open*, indicating that the case it

```
do-assumptive-projection( c )
  begin
     if worth-pursuing( c ) then
     begin
        E := e-elements-with-concession( c );
        for each e ∈ E
        begin
           c' := choose-successor( assumptive-occurrence( e, c ) );
           add-arc-to-projection( c, c' );
           if status( c' ) = open then do-assumptive-projection( c' )
        end
     end
     else set-status-to-closed( c )
  end.
```

Figure 6.15: The algorithm for doing an assumptive projection.

contains merits future exploration.

While this algorithm is not really "smarter" than STRIPS at finding correct plans, it will often do significantly less work, due to the fact that the plan nets it constructs allow so much work to be saved across branches of the state-space used to control the choices open during plan construction.

Chapter 7: Suggestions for Future Research.

7.1. Chapter overview.

Various suggestions have been made throughout the previous chapters on areas in which the current work requires extension and refinement. This chapter represents an attempt to collect these suggestions together. No apologies are made for half-baked ideas, since we present suggestions, not final work. Precision is sacrificed in the hope of generating inspiration.

7.2. More execution advice.

We have viewed a plan net's execution advice as a means for preventing the occurrence of causally permitted, but unintended actions. The only sort of advice we have defined is the *before* relation, used to select between equally attractive alternatives for execution. Given a plan's projection, one can view this before order as defining a set of paths through the projection, from the initial to goal case. It seems possible to extend this notion of execution advice to other sorts of projection navigation. In particular, some execution advice regarding loop termination seems desirable. What would be required is information about when to stop repeating the execution of any given action. This information could be provided in a plan net $P = (B,E;F,A)$, by making the execution advice, A, an n-tuple, and letting one of the components of the tuple be the existing *before* relation. One of the new components could be a set of pairs of e-elements and b-elements. Call this new component U, for *until*. Then $U \subseteq (E \times B)$. The intended meaning of U would be that for each ordered pair, $(e,b) \in U$, if e has concession in some case c, it will be considered for execution only if $\{b\} \cap c = \varnothing$. This would effectively capture the notion of *e until b*, and would seem to be useful for terminating many iterative plan net constructs. The difficulty arises in being precise about how the until advice, U, enters into the construction of a plan net's projection. This may or may not be difficult, and requires more thought.

7.3. Distinguishing between sensory and inferential actions.

We have suggested at various points that the plan net formalism can represent both sensory and inferential actions, but that it cannot distinguish between them. It would be useful to have some

sort of understanding of the differences between sensory and inferential actions, so that initial steps towards a formalisation could be taken. Almost certainly, something about this exists in the philosophical literature. Someone should attempt to operationalise existing philosophical thought on this subject in the context of automated planning.

7.4. System accountability.

An oft touted advantage of "expert systems" is their ability to explain their behaviour. It might be interesting to use the plan net framework to explore this notion of explanation. If explanations are given by recourse to inferential justifications, then plan nets offer a great deal of relevant information. By using the ordering relations in a plan net, it would seem possible to produce extremely well structured accounts of how the system has produced a certain conclusion. Using some of our definitions, it should also prove possible for the system to occasionally admit that it has actually *assumed* something, and that it does not in fact *believe* it.

7.5. Action hierarchy.

We have totally ignored the issue of action hierarchy. This is one of the major results in automatic planning over the last few years, and it is necessary to integrate some ideas regarding action hierarchy in the plan net framework. We have not done so, simply to constrain the nature of the task at hand. With the basic plan net framework now in place, it appears possible to formalise various notions of action hierarchy. The currently available ideas regarding this topic are by no means consistent with one another; plan nets should provide an extremely well structured and formal framework for exploring the utility of the various ideas in the literature. Recent work by Tenenberg (1986) looks promising.

7.6. Use of a black-board architecture.

We have not discussed the issue of plan generation in any depth, so it is hardly surprising that plan generation system *architectures* have not been touched on at all. Some modern planners, such as O-Plan (Currie & Tate, 1985), have an agenda of pending tasks, and implement an

executive which sees to task achievement by invoking appropriate knowledge sources as required. Such an architecture also looks promising for the automatic generation of plan nets. The blackboard would probably contain the developing plan, the domain constraints, and the plan net's (assumptive) projection. Such an arrangement would facilitate plan execution and progress monitoring, since the various algorithms presented in chapter 4 could be implemented as knowledge sources operating over the globally available data structures.

7.7. Retrojections.

We have defined plan net projections as structures which extend from some initial case using the definition of e-element occurrence to build an accessibility relation on cases. Some modern planners do not seem to operate this way. Instead, they build a plan "from right to left"; in some sense, backwards in time. The plan is constructed from the goals to the initial situation. A plan net projection does not seem to adequately capture the sort of reasoning that would be going on during such construction. What is required is some idea of a plan net *retrojection*, or inverse projection, going from the goal case to possible initial cases. This structure would seem useful for formalising the sort of plan construction reasoning embodied in some systems.

A plan net retrojection may also prove useful in other situations. Consider the scenario in which an agent is given the final and initial conditions of some process, and asked to come up with a specification of what the process might have been that transformed the initial into the final. Such a problem might be termed a *rational reconstruction*, or a "how did we get here from there" sort of problem. The situation is slightly different to that found in planning problems, where the agent is given an initial situation specification, and a set of goals, and required to come up with a plan by which the initial situation may be turned into one which satisfies the goal specification. The rational reconstruction problem is similar, but slightly different in that the process has already taken place: the agent must describe what has *already* happened, and not what might happen.

7.8. Plan net construction.

New ideas on plan net generation are badly needed. It seems reasonably straightforward to generate basic plan nets, as demonstrated by the simple algorithm of chapter 6. And it shouldn't prove difficult to build planners which generate plan nets using much the same techniques as modern day systems. However, what we really need are some ideas regarding the generation of iterative plans. In this area, ideas from program synthesis may prove useful. Work on programming by invoking program "templates" should be relevant. (See for instance, Rich, 1981, for some pointers into the area.)

7.9. Dependency directed backtracking.

The definition of the belief consistency maintenance function, μ, is too simplistic. It must be defined to take into account the arguments by which the detected inconsistencies were brought into being. In this way, inconsistencies could be traced down to their source, and measures taken to see that they do not recur. The need for some sort of plan net *retrojection* seems to come in here: by doing a net retrojection, it may be possible to recreate the cases through which the argument progressed, and thus come to the base set of assumptions used to start the argument. Exactly how this must be done isn't clear, but there are some interesting prospects.

A related point is that by removing certain formulas of L_B, we may be invalidating the argument for some other current belief. Thus, set subtraction in the definition of assumptive (or realistic) follower case is inadequate. Those beliefs which depend on the removed formulas for being *in* must also be found and removed. By directly removing the inconsistencies found by μ, we may punch holes in existing arguments, and must find and force out these arguments' results. More recent work by de Kleer (1986a) may prove relevant.

7.10. Representing conditional proofs.

We have not included anything equivalent to Doyle's (1978) Conditional Proof justification structure. Whether or not this is a problem remains to be seen. If it is a problem, then a fix is

required. How to do this isn't clear.

7.11. Which projection to use?

We have presented two sorts of plan net projection: assumptive and realistic. When should one be used over the other? We have made initial suggestions in this area, but nothing has yet been formalised. In fact, it is possible to mix assumptive and realistic steps within the *same* projection. A method of operator application might be selected on the basis of the nature of the actions being modelled by the e-elements in the step. In this connection, it might be interesting to try to include some ideas regarding action utility and probability of action outcome. Work by Doyle, Atkinson, and Doshi (1986) is beginning to consider *when* assertions within a plan require monitoring.

7.12. Possible search spaces.

While we have spent some time describing the relationship between a search space of possible world states and partial plans, it seems that still more could be said. In particular, it would be nice to understand more precisely the relationship between doing search through partial plans and the ability to use a partial plan as a record of the search already done. The plan construction technique we presented used a plan net to cache the work it had done, so that backtracking through the world state space did not mean throwing away results. Perhaps if a plan representation is expressive enough, then it actually can perform *all* the work required of a reason maintenance facility. We have taken some initial steps towards showing that this is so, but much more work is required, especially of a formal nature.

7.13. Typed preconditions.

Some modern planners have introduced the notion of *typed preconditions*. In particular, NonLin (Tate, 1977, 1984), O-Plan (Currie & Tate, 1985), and SIPE all include some version of the idea. Essentially, there are some "preconditions" for actions which should never be considered as goals. For instance, there may be an abstract action for building houses of less than three floors. We might consider the condition "house is less than 3 floors" as a precondition of the abstract action.

But we would not want to plan to achieve this condition. Rather, it is to be viewed as an applicability check on the operator. If the house being constructed *is* less than 3 floors, then the operator is relevant. If not, then the operator should not be considered. At no time would we consider changing the nature of the problem to suit the operator's preconditions, so in the context of this example, we should not consider altering the height of the structure we are building. Such an idea must somehow be added to the plan net framework, or a demonstration given of the framework's ability to already handle such preconditions.

7.14. Plan execution monitoring.

Renewed interest is being demonstrated in plan execution monitoring (Tate, 1984a; Wilkins, 1985). The success of plan execution monitoring depends crucially on the plan representation's ability to capture teleological information. This is because what constitutes a plan failure depends on what the declared purpose of the plan is; the purpose of a plan determines its *reasons* for being. For example consider a plan which during its execution calls for block *a* to be moved from block *b* to block *c*. Imagine that somehow, after *a* is lifted from *b*, another block, *d*, is placed on *b*. Has the plan failed? The answer to this question depends on what the plan's purpose is. If the plan was to get block *b* on top of block *c*, then it has been a success. If, however, the plan was to get block *a* to be clear, then it has failed. The reasoning which is to determine whether or not a plan has failed must therefore have access to a plan's *reasons* for being. Such reasons are captured in a plan net's ordering relations. An interesting research project would be to examine the utility of this information in doing effective plan execution monitoring.

7.15. The nature of iteration.

We have not been very precise about the sort of iteration that plan nets can describe. The iterative examples of chapter 5 involve only cyclic processes, in which the *same event* appears to happen many times. For instance, the plan which included a specification of a traffic light's colour alternation included two events, one for each of the colour changes. Consider some event in which the light changes from *red* to *green*. There is nothing to distinguish this one *instance* of a

colour change from any other, and that is precisely why plan nets work. We do not distinguish one particular event instance from another, and thus are allowed to include a single "generic" event specification in a plan net, and have it stand for all possible instances.

This trick will not work for all sorts of iteration. Consider the situation in which an action is applied to all elements of a given set. Say the problem is to hit each nail in a given set of nails exactly once. It would be nice to have a single event specification, *hit(n)*, and allow *n* to range over the given set of nails. This is asking for some version of bounded quantification, wherein a variable is allowed to range over a given set. Each of the possible bindings for the variable would be used to create an action instance. So far, the plan net framework has no means for specifying this sort of iteration through bounded quantification.

Related to the issue of iteration is the extension of the execution advice to include an *until* specification, as suggested above. An *until* construct would not seem to increase the range of iteration that can be described, but would certainly make termination easier (and that can't be a bad thing).

7.16. Keeping track of current reasons.

As an agent operates, it will *surely* acquire new and *unplanned* beliefs. The definitions of chapter 4 make no provision for this. The problem is that our definition of belief is made with respect to a given plan, viewed as a set of reasons. Now while an agent might believe something, it may not have any reasons, in the sense we have defined, unless it keeps track of where unplanned beliefs arise from. The current plan can't be used to justify all beliefs, since it might not include a specification of the actions by which the beliefs have been created. This demonstrates the need for a "reason maintainer", which is charged with keeping the agent's current set of reasons up to date with its experience. Once this set of reasons exists, the definitions we have given would seem to be adequate. Of course, this issue requires more thought than it has been given in this thesis.

7.17. Importing more net theory ideas.

There are many analysis tools within Net Theory that should prove useful for dealing with plan nets. An excellent introduction to Net Theory can be found in Reisig (1985). Some ideas that seem relevant are those involving synchronic distance, and the use of occurrence nets to analyse possible plan net behaviour. Plan net extensions which allow multiple tokens (i.e., cases are allowed to be *bags*, not just sets) appear interesting, if problematic.

Chapter 8: Conclusions.

The struggle for existence holds as much in the intellectual as in the physical world. A theory is a species of thinking, and its right to exist is coextensive with its power of resisting extinction by its rivals.

– T. H. Huxley

8.1. Chapter overview.

This chapter itemises the contributions of the thesis presented in this dissertation. A summary of the dissertation and the structure of our argument is not provided, since the introduction includes this information. We break down the general notion of contribution into two parts: technical results, and lessons that have been learned along the way. The title of this second section does not refer to lessons learned only by the author; a reader should have learned them also, provided the author has done his job. This chapter is intentionally short, to enable quick access to the major results of the thesis.

8.2. Technical results.

We have formally defined an expressive and flexible representation for plans, suitable for use in automatic planning systems. The representation builds on previous work in automatic planning, and incorporates many desirable features of existing representations. The key features of our representation include the following.

Precision

Plan nets have been formally defined using simple set theory and net theory tools, and are thus independent of any particular implementation as a part of a computer program. This represents an advance over most of the plan structures to be found in the automatic planning literature (the notable exceptions being Georgeff, 1984; Chapman, 1985; and Pednault, 1985).

How action affects belief

Plan nets attempt to characterise the changes to an agent's beliefs under the occurrence of an action. No previous work has attempted to do this, as far as the author is aware. Plan nets are very general, and can be used to describe the changes to an agent's beliefs brought on by the occurrence of *any* given action. Because of this, plan nets are able to model "sensory" actions; that is, actions by which the agent is made aware of some property of its surrounding environment.

Iteration in an action-ordering structure

Plan nets can describe certain sorts of iterative behaviour. In particular, cyclic processes involving event re-occurrence are easily modelled using plan nets. Previous action-ordering representations have been unable to do this.

Teleology

Teleology is the basic "glue" from which plan nets are built. This represents a shift from classical action-ordering plan representations, in which the major ordering relation is interpreted as *before*. It is this focus on plan teleology that permits plan nets to describe iterative behaviour.

Agent intention vs. causal possibility

Plan nets responsibly distinguish those orderings on actions enforced by teleology and those enforced by agent preference. The first relates to causal possibility; the latter, to intention. In the plan net framework, the distinction is captured by the different ordering relations in a plan net tuple. Teleological relationships among actions are used to define plan net structure, and agent-preferred *before* relations are encoded in a plan's execution advice. We have shown how it is possible to view an agent's preferences regarding before-ness on actions as defining a route through a state-space graph of possible world states. We have formalised this in the definition of *sound* advice. We call an agent's set of *before* preferences sound if and only if they can be expected constrain what is causally possible to what is desired by the agent.

Simple plan analysis

As part of the plan net framework, we have provided implementation independent plan analysis tools, in the form of assumptive and realistic plan net projections. These projections are state-space pictures of what a plan net can do. An assumptive projection is very similar to the state-space searched by systems such as STRIPS. However, a system using plan nets does not need to always *search* through the state space: portions of the space can be constructed on demand, with the plan net being used to record the actions and ordering

relations required to traverse the space.

Causal independence and parallelism

Both the basic plan net structures and projection analysis tools allow for causally independent actions. This means that plan nets can represent, and projections can help reason about, parallel plans. Such plans are necessary for dealing with least commitment action-ordering strategies. We have also clearly defined when two actions are causally independent.

Action disjunction

Plan nets can easily represent disjunction. The disjunction represented in a plan net becomes non-deterministic choice in the projection analysis.

External actions

We introduced a concept called *event executancy*, and have used it to argue that plan nets are able to model actions which are completely external to an agent. Executancy only becomes relevant when an agent attempts to realise the behaviour a plan net specifies.

Harmony with reason maintenance

A plan net's projection performs some of the work required of a reason maintenance system. It now seems that a planning system can avoid wasted work by storing its results in a plan net: the net can be used to create portions of the search space on demand. While more work is obviously required in this area, this thesis has shown some of the basic relationships between plan nets and Doyle-like reason maintenance systems; in particular, we have explained the relationship between reachability in a plan net's projection, and strict groundedness in the sense defined by Doyle.

8.3. Lessons learned.

Plan nets have been motivated in two ways. In chapter 2, we motivated how they characterise the changes to an agent's beliefs under action occurrence. In chapter 3, we argued for the way in which they use teleological orderings to build plans out of individual event descriptions. During

the course of these two chapters, a couple of issues were discussed which are worth remembering.

The linearity assumption and parallel plans

Parallel plans do not "solve" the linearity assumption. The linearity assumption involves plan construction; parallel plans are plans in which there is potential concurrency of action.

Partial plans and reason maintenance

There is an interesting relationship between a problem solving architecture which calls for search through partial plans, and the use of a plan net structure to record the dependencies among planned actions. It begins to look as though search through partial plans, and the ability to do reason maintenance go hand in hand, at least in a planning context. Of course, more work is required to say anything definitive about this.

References.

Allen, J. 1981. An interval-based representation of temporal knowledge. *Proc of the 7th IJCAI.* Vancouver, B.C., Canada.

Allen, J., and Koomen, J. 1983. Planning using a temporal world model. *Proc. of the 8th IJCAI.* Karlsruhe, West Germany. pp. 741-747.

Appelt, D. 1985. *Artificial Intelligence.* Vol. 26, pp. 1-33.

Barnden, J. 1986. A viewpoint distinction in the representation of propositional attitudes. In the proc. of *AAAI-86.* Philadelphia, PA. pp. 411-415.

Barr, A. and Feigenbaum, E. 1982. *The handbook of artificial intelligence.* Volume III. William Kaufmann Inc.

Brand, M. 1984. *Intending and acting: towards a naturalized action theory.* Bradford books, MIT press. Cambridge, MA.

Brauer, W. (ed.). 1979. Springer-Verlag LNCS series. *Net theory and applications.* Proceedings of the Advanced Course on General Net Theory of Processes and Systems, Hamburg.

Bundy, A. 1977. Exploiting the properties of functions to control search. University of Edinburgh, Department of Artificial Intelligence Research Report No. 45.

Chapman, D. 1985. Nonlinear planning: a rigorous reconstruction. In proc. of *IJCAI-9.* pp. 1022-1024.

Charniak, E., and McDermott, D. 1985. *Introduction to Artificial Intelligence.* Addison-Wesley.

Charniak, E., Riesbeck, C., and McDermott, D. 1979. *Artificial intelligence programming.* Lawrence Erlbaum, Hillsdale, NJ.

Cheeseman, P. 1983. A representation of time for planning. Tech. note 278, SRI International, Menlo Park, CA.

Currie, K., & Tate, A. 1985. O-Plan: Control in the open planning architecture. *Proc. of the BCS Expert Systems '85 Conference.* Warwick (December), Cambridge University Press.

Czaja, L. 1983. Making nets abstract and structured. Oxford University Computing Laboratory, Programming Research Group, 8-11 Keble Road, Oxford, U.K.

Davis, L. 1979. *Theory of action.* Englewood cliffs, NJ. Prentice-Hall.

de Kleer, J. 1984. Choices without backtracking. In proc. of *AAAI-84.* pp. 79-85.

de Kleer, J. 1986a. An assumption-based truth maintenance system. *Artificial Intelligence.* Vol. 28, pp. 127-162.

de Kleer, J. 1986b. Extending the ATMS. *Artificial Intelligence.* Vol. 28, pp. 163-196.

de Kleer, J. 1986c. Problem solving with the ATMS. *Artificial Intelligence.* Vol. 28, pp. 197-224.

de Kleer, J., and Williams, B. 1986. Back to backtracking: controlling the ATMS. In proc. of *AAAI-86.* Philadelphia, PA. pp. 910-917.

Doyle, J. 1979. A truth maintenance system. *Artificial Intelligence.* Vol. 12, pp. 231-272.

Doyle, J. 1982. Some theories of reasoned assumptions: an essay in rational psychology. Department of Computer Science, Carnegie-Mellon University. (CMU-CS-83-125).

Doyle, J. 1983. The ins and outs of reason maintenance. In proc. of *IJCAI-8.* pp. 349-351.

Doyle, J. 1985. Reasoned assumptions and pareto optimality. In proc. of *IJCAI-9.* pp. 87-90.

Doyle, R.J., Atkinson, D.J., and Doshi, R.S. 1986. Generating perception requests and expectations to verify the execution of plans. In proc. of *AAAI-86.* Philadelphia, PA.

Drummond, M.E. 1985a. Non-linear planning in terms of cause and effect. In proc. of SCS multi-conference *AI, graphics, and simulation.* San Diego, CA.

Drummond, M.E. 1985b. Refining and extending the procedural net. In proc. of *IJCAI-9.* pp. 1010-1012.

Drummond, M.E. 1986. A representation of action and belief for automatic planning systems. In proc. of CSLI/AAAI workshop on Planning & Action, Oregon, U.S.A. (Also Artificial Intelligence Applications Institute technical report AIAI-TR-16.)

Ernst, G., and Newell, A. 1969. *GPS: a case study in generality and problem solving.* Academic Press, New York.

Fikes, R.E., Nilsson, N.J. 1971. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2.* pp. 189-208.

Fodor, J. 1981. Methodological solipsism considered as a research strategy in cognitive psychology. In John Haugeland (ed.). *Mind Design.* Montgomery, Vt. Bradford Books.

Georgeff, M. 1984. A theory of action for multiagent planning. In proc. of *AAAI-84.* Austin, TX. pp. 121-125.

Georgeff, M., Lansky, A., Bessiere, P. 1985. A procedural logic. In proc. of *IJCAI-9.* pp. 516-523.

Gray, P. 1984. *Logic, algebra and databases.* Ellis Horwood, Chichester.

Green, C.C. 1969. Theorem-proving by resolution as a basis for question-answering systems, In Machine Intelligence, Vol. 4, B. Meltzer and D. Michie (eds.). American Elsevier Publishing Co., New York.

Haas, A.R. 1986. A syntactic theory of belief and action. *Artificial Intelligence.* Vol. 28. pp. 245-292.

Hayes-Roth, B., and Hayes-Roth, F. 1979 A cognitive model of planning. *Cognitive science,* pp. 275-310.

Hayes, P. 1981. The frame problem and related problems in artificial intelligence. In *Readings in artificial intelligence.* B.L. Webber & N.J. Nilsson (eds.), Tioga publishing co., Palo Alto, CA. pp. 223-230. Originally appeared in *Artificial and human thinking.* A. Elithorn & D. Jones (eds.), Jossey-Bass. pp. 45-59.

Hewitt, C. 1972. Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot. Ph.D. thesis, Department of Mathematics, MIT, Cambridge, MA.

Hintikka, J. 1962. *Knowledge and belief.* Ithica, NY: Cornell University Press.

Konolige, K. 1980. A first order formalization of knowledge and action for a multiagent planning system. Artificial Intelligence Center Technical Note 232, SRI International, Menlo Park, California. (Also in *Machine intelligence 10*, 1981).

Kowalski, R., & Sergot, M. 1984. A logic-based calculus of events. Alvey IKBS planning special interest group, 3rd Planning workshop, Sunningdale, U.K. (Avail. from IEE, London.).

Lansky, A. 1985. Behavioral specification and planning for multiagent domains. Tech. note 360, SRI International, Menlo Park, CA.

Levesque, H. 1984. A logic of implicit and explicit belief. *Proc. of AAAI-84.* Austin, Texas (August).

McCarthy, J. 1958. Programs with common sense. *Semantic information processing.* M. Minsky, ed., pp. 403-418. Cambridge: MIT Press.

McCarthy, J., and Hayes, P.J. 1969. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence 4.* B. Meltzer & D. Michie, eds. pp. 463-502. Edinburgh: Edinburgh University Press.

McDermott, D. 1983. Generalizing problem reduction: a logical analysis. *Proceedings of IJCAI-83.* pp. 302-308.

Miller, G.A., Galanter, E., and Pribram, K.H. 1960. *Plans and the structure of behavior.* Henry Holt and Company, New York.

Moore, R.C. 1977. Reasoning about knowledge and action. In proc. of *IJCAI-5.* pp. 223-227.

Moore, R.C. 1980. Reasoning about knowledge and action. Artificial Intelligence Center Technical Note 191, SRI International, Menlo Park, California.

Moore, R.C. 1985. A formal theory of knowledge and action. In J.R. Hobbs & R.C. Moore (eds.) *Formal theories of the commonsense world.* Ablex series in artificial intelligence.

Moore, R. C. and Hendrix, G. 1982. Computational models of belief and the semantics of belief sentences. In S. Peters & E. Saarinen (eds.). *Processes, beliefs, and questions.* D. Reidel Publishing Co. pp. 107-127.

Moore, R.C. 1985. Semantical considerations on non-monotonic logic. *Artificial Intelligence.* Vol. 25, pp. 75-94.

Nielsen, M., and Thiagarajan, P.S. 1984. Degrees of non-determinism and concurrency: a petri net view. *In proc. of the 4th conference on foundations of software technology and theoretical computer science.* (December).

Nilsson, N. 1980. *Principles of Artificial Intelligence.* Tioga Publishing Co. Palo Alto, CA.

Nilsson, N. 1985. Triangle tables: a proposal for a robot programming language. Artificial Intelligence center Technical Note 347, SRI International, Menlo Park, CA.

Pednault, E.P.D. 1985. Preliminary report on a theory of plan synthesis. Artificial Intelligence Center Technical Note 358, SRI International, Menlo Park, CA.

Peterson, J.L. 1977. Petri nets. *Computing surveys.* Vol. 9, No. 3, September.

Peterson, J.L. 1980. *Petri net theory and the modeling of systems.* Prentice-Hall.

Pippenger, N. 1980. Pebbling. RC 8258 (#35937), IBM TJ Watson Research Center, Yorktown Heights, NY.

Reiter, R. 1978. On closed world data bases. In *Logic and data bases.* H Gallaire & J. Minker (eds.), Plenum Press, New York.

Quine, W.V. 1953. Two dogmas of empiricism. In. *From a logical point of view.* Harvard University press, Cambridge.

Quine, W.V., and Ullian, J.S. 1978. *The web of belief.* Random House, NY.

Reisig, W. 1985. *Petri nets: an introduction.* Springer-Verlag, EATCS Monographs on theoretical computer science, vol. 4.

Rich, C. 1981. A formal representation for plans in the programmer's apprentice. In proc. of *IJCAI-81.*

Sacerdoti, E.D. 1975a. A structure for plans and behaviour. Stanford Research Institute Technical Note #109.

Sacerdoti, E.D. 1975b. The non-linear nature of plans. In *Proc. of IJCAI-75.*

Sacerdoti, E.D. 1977. *A structure for plans and behavior.* American Elsevier, New York. Also SRI AI Tech. Note 109, August, 1975.

Sacerdoti, E. 1979. Problem solving tactics. *Proceedings of IJCAI-79.* pp. 1077-1085.

Sacerdoti, E.D. 1980. Plan generation and execution for robots. Stanford Research Institute TN-209.

Simon, H.A. 1983. Search and reasoning in problem solving. *Artificial Intelligence.* No. 21. pp 7-29.

Sridharan, A., and Bresina, A. 1982. Plan formation in large, realistic domains. In proc. of 4th Biennial conf. of Canadian Society for Computational Studies of Intelligence (CSCSI), Saskatoon, Sask.

Stefik, M. 1981. Planning with constraints (Molgen: Part I), *Artificial Intelligence.* Vol. 16, pp. 111-140.

Sussman, G.J., Winograd, T., & Charniak, E. 1971. Micro-planner reference manual. MIT Tech. Rept. 203a.

Tarski, Alfred. 1944. The semantic conception of truth. *Philosophy and phenom. research.* Vol. 4, pp. 341 ff.

Sloman, A. 1985. Real time multiple-motive expert systems. *Proc. of the BCS Expert Systems '85 Conference.* Warwick (December), Cambridge University Press.

Tate, A. 1974. Interplan: a plan generation system which can deal with interactions between goals. University of Edinburgh, Machine Intelligence Research Unit Memorandum MIP-R-109.

Tate, A. 1975a. Interacting goals and their use. *Advance papers of the 4th IJCAI, Tbilisi, U.S.S.R.*

Tate, A. 1975b. *Using goal structure to direct search in a problem solver.* Ph.D. thesis, University of Edinburgh, Edinburgh, Scotland.

Tate, A. 1976. Project planning using a hierarchic non-linear planner. University of Edinburgh, Department of Artificial Intelligence Research Report #25.

Tate, A. 1977. Generating project networks. *Proceedings of the 5th IJCAI.*

Tate, A. 1984a. Planning and condition monitoring in an FMS. International conference on Flexible Automation Systems. Institute of Electrical Engineers, London, UK (July).

Tate, A. 1984b. Goal structure -- capturing the intent of plans. *Proceedings of ECAI-84.* (September).

Tate, A. 1985. A review of knowledge-based planning techniques. In proc. of *Expert systems 85.* (5th Technical conference of the British Computer Society, Specialist group on expert systems.) University of Warwick. pp. 89-111.

Tenenberg, J. 1986. Planning with abstraction. In proc. of *AAAI-86.* Philadelphia, PA. pp. 76-80.

Vere, S.A. 1981. Planning in time: windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence.* Vol. PAMI-5, No. 3, pp. 246-267.

Vere, S.A. 1985. Splicing plans to achieve misordered goals. In proc. of *IJCAI-9.* pp. 1016-1021.

Waldinger, R. 1977. Achieving several goals simultaneously. *Machine Intelligence 8.* In E.W. Elcock and D. Michie (Eds.), New York, Halstead/Wiley.

Warren, D.H.D. 1974. Warplan: a system for generating plans. Memo 76, Computational Logic Dept., School of Artificial Intelligence, University of Edinburgh.

Wegner, P. 1970. Programming language semantics. *Formal semantics of programming languages.* R. Rustin, ed. Prentice-Hall, Inc., Englewood Cliffs, NJ. pp. 149-248.

Wilkins, D.E. 1983. Representation in a domain independent planner. In proc. of *IJCAI-8.* pp. 733-740.

Wilkins, D.E. 1984. Domain independent planning: representation and plan generation. *Artificial Intelligence,* No. 22.

Wilkins, D. 1985. Recovering from execution errors in SIPE. In *Computational Intelligence 1.* pp. 33-45.

Appendix A:
Prolog Implementation of Belief Consistency Maintenance.

```
/*
|       File: mu-x.t
|       Contains: Prolog code for belief consistency maintenance.
|       Updated: Sat Aug  2 18:45:28 1986
|
|       -- Mark Drummond
*/



/*
|       First the utilities needed to make life simpler
*/



% For administration

1 :-
        reconsult(' mu-x.t  ').



% Set union of 1st and 2nd args gives 3rd.  4th arg says whether to attempt
% bindings for variables in making the set union.

union(S1,[],S1,F) :-
        !.
union(S1,[El|S2],S3,F) :-
        element(El,S1,F),
        !,
        union(S1,S2,S3,F).
union(S1,[El|S2],[El|S3],F) :-
        union(S1,S2,S3,F).



% Set intersection of 1st and 2nd args gives 3rd.  4th arg says whether to
% attempt bindings in making set intersection.

intersection(S1,[],[],F) :-
        !.
intersection(S1,[El|S2],[El|S3],F) :-
        element(El,S1,F),
        !,
        intersection(S1,S2,S3,F).
intersection(S1,[_|S2],S3,F) :-
        intersection(S1,S2,S3,F).
```

```
% Succeed if 1st arg is in the 2nd.  3rd arg indicates if binding is desired
% for variables.



element(El1,[El2|S2],F) :-
        matchable(El1,El2,F).
element(El,[_|S],F) :-
        element(El,S,F).



% Succeed if 1st 2 args are "equal", and use the 3rd arg to determine the
% method for testing equality.

matchable(S,S,match).
matchable(S1,S2,nomatch) :-
        S1 == S2.
matchable(S1,S2,F) :-
        set_equal(S1,S2,F).



% Succeed if 1st and 2nd args (sets) are the same set.

set_equal(S1,S2,F) :-
        set_diff(S1,S2,[],F),
        set_diff(S2,S1,[],F).



% 3rd arg is the 1st minus the 2nd (set subtraction), and the 4th arg indicates
% whether or not binding is desired for variables.

set_diff(S,[],S,_) :-
        !.
set_diff(S1,[El|S2],Ans,F) :-
        remove(El,S1,S3,F),
        set_diff(S3,S2,Ans,F).
set_diff(S1,[_|S2],Ans,F) :-
        set_diff(S1,S2,Ans,F).
```

```
% 3rd arg is the 2nd without the 1st. 4th arg determines how equality is
% tested.



remove(El,[El|S],S,match).
remove(El1,[El2|S],S,nomatch) :-
        El1 == El2,
        !.
remove(El1,[El2|S],[El2|Ans],F) :-
        remove(El1,S,Ans,F).



/*
| det_poss_incon: determine possible inconsistencies.
|       1st arg: A tag to find the appropriate constraint set. (Entry bound)
|       2nd arg: New observations; a set of terms. (Entry bound)
|       3rd arg: Constraints that might be violated. (Exit bound)
|
| Assumes dcs(Tag,D) exists, and uses it to access required domain
| constraints.
*/

det_poss_incon(Tag,[],[]) :-
        !.
det_poss_incon(Tag,[New|More],All) :-
        dcs(Tag,Constr),
        examine_constrs(New,Constr,Some),
        det_poss_incon(Tag,More,Result),
        union(Some,Result,All,nomatch).



examine_constrs(New,[],[]) :-
        !.
examine_constrs(New,[Constr|More],Ans) :-
        assert(bindings([])),
        match_one_constr(New,Constr),
        retract(bindings(Result)),
        !,
        examine_constrs(New,More,Rest),
        union(Result,Rest,Ans,nomatch).



match_one_constr(New,Constr) :-
        element(New,Constr,match),
        side_effect(Constr),
        fail.
match_one_constr(_,_).
```

```
/*
| del_new_obs: delete new observations.
|         1st arg: a set of domain constraints. (Entry bound)
|         2nd arg: a set of new observations. (Entry bound)
|         3rd arg: 1st arg with all elements containing any element of 2nd arg
|                 removed. (Exit bound)
*/

del_new_obs(Constrs,[],Constrs) :-
        !.
del_new_obs(Constrs,[New|More],Clean) :-
        del_single_ob(Constrs,New,Fresh),
        del_new_obs(Fresh,More,Clean).



del_single_ob([],_,[]) :-
        !.
del_single_ob([Constr|More],New,[Fresh|Rest]) :-
        element(New,Constr,nomatch),
        !,
        set_diff(Constr,[New],Fresh,nomatch),
        del_single_ob(More,New,Rest).
del_single_ob([Constr|More],New,[Constr|Rest]) :-
        del_single_ob(More,New,Rest).



/*
| factor: factor constraint sets into sets of possible deletions.
|         1st arg: A set of domain constraints. (Entry bound)
|         2nd arg: A set of possible deletions. (Exit bound)
|
| Example: factor([[a,b,c],[c,d]],[[a,d],[a,c],[b,d],[b,c],[c,d],[c]]).
*/

factor([],[[]]) :-
        !.
factor([M|Tail],Deletions) :-
        factor(Tail,R),
        for_each_m(M,R,Deletions).



for_each_m([],_,[]) :-
        !.
for_each_m([Front|M],R,Deletions) :-
        for_each_r(Front,R,Nr),
        for_each_m(M,R,Ans),
        union(Ans,Nr,Deletions,nomatch).
```

```
for_each_r(_,[],[]) :-
        !.
for_each_r(Front,[Rest|R],Out) :-
        union(Rest,[Front],Res,nomatch),
        for_each_r(Front,R,Ans),
        union([Res],Ans,Out,nomatch).


% Sort a list using size of its component lists.

order_by_size([],[]) :-
        !.
order_by_size([El|Rest],[El|Result]) :-
        size(El,N),
        smallest(N,Rest),
        !,
        order_by_size(Rest,Result).
order_by_size([El|Rest],Result) :-
        order_by_size(Rest,Ans),
        insert(El,Ans,Result).


% Insert 1st arg in set which is second at location appropriate to its size.

insert(S,[],[S]) :-
        !.
insert(S1,[S2|More],[S2|Ans]) :-
        size(S1,N),
        size(S2,M),
        N > M,
        !,
        insert(S1,More,Ans).
insert(S1,S2,[S1|S2]).


% Find size of first arg, set second to it

size([],0) :-
        !.
size([El|S],N) :-
        size(S,N1),
        N is N1 + 1.


% Succeed if first arg is size of smallest set in second arg

smallest(_,[]) :-
        !.
smallest(N,[Set|Rest]) :-
```

```
        size(Set,M),
        N < M,
        smallest(N,Rest).


% Choose one element from a set, fail when no more (could use some heuristic to
% guide choice, instead of just selecting the first one found).



choose_one(El,[El|_]).
choose_one(El,[_|More]) :-
        choose_one(El,More).



/*
| can_be_bound: finds all possible bindings for a set of domain constraints
|                   in a set of beliefs.
|         1st arg: A set of domain constraints. (Entry bound)
|         2nd arg: A set of beliefs. (Entry bound)
|         3rd arg: A set of domain constraints, derived from 1st arg, using all
|                   possible combinations of constants found in 2nd arg.
*/

can_be_bound([],_,[]) :-
        !.
can_be_bound([One|More],Bels,Result) :-
        assert(bindings([])),
        all_bindings(One,Bels),
        retract(bindings(Some)),
        can_be_bound(More,Bels,Ans),
        union(Some,Ans,Result,nomatch).



all_bindings([],_) :-
        !.
all_bindings(One,Bels) :-
        set_diff(One,Bels,[],match),
        side_effect(One),
        fail.
all_bindings(_,_).



% This is ugly, but Prolog leaves little choice!

side_effect(One) :-
        retract(bindings(B)),
        union(B,[One],U,nomatch),
```

```
        assert(bindings(U)),
        !.


/*
| mu: Belief consistency maintenance function
|       1st arg: A tag to find a set of domain constraints. (Entry bound)
|       2nd arg: A set of current beliefs. (Entry bound)
|       3rd arg: A set of new observations. (Entry bound)
|       4th arg: A set of possible deletions. Each possible deletion is a set,
|                   and the result is ordered so that the smallest deletions
|                   appear first. (Exit bound)
*/


mu(Tag,Beliefs,Observations,Result) :-
        det_poss_incon(Tag,Observations,Possible),
        del_new_obs(Possible,Observations,Constraints),
        set_diff(Beliefs,Observations,OnlyOld,nomatch),
        can_be_bound(Constraints,OnlyOld,Deletions),
        factor(Deletions,Result).


/*
| survive: top level demonstration loop which uses "mu" to maintain belief
|           consistency.  Reads in new observations, and updates beliefs.
*/

survive(Beliefs,Tag) :-
        write(' Current beliefs are:'),
        nl,
        write('  '),
        write(Beliefs),
        nl, nl,
        prompt(_,' New observations? '),
        read(Observations),
        Observations \= q,
        !,
        mu(Tag,Beliefs,Observations,Choices),
        order_by_size(Choices,Result),          % A very simple ψ!
        write(' Choices: '),
        write(Result),
        nl,
        choose_one(One,Result),
        write(' Choosing to disbelieve: '),
        write(One),
        nl,
        set_diff(Beliefs,One,Cleaned,nomatch),
        union(Cleaned,Observations,Updated,nomatch),
        survive(Updated,Tag).
```

```
survive(_,_) :-
        write(' Finished '),
        nl.



/*
| bw: a blocks-world demonstrator. Set up initial beliefs, and survive, using
|     blocks world domain constraints.
*/

bw :-
        Beliefs = [clr(a),
                   clr(t),
                   on(a,b),
                     on(b,t),
                     block(b),
                     block(a),
                     table(t)],
        survive(Beliefs,bwc).



/*
| bw1: another blocks-world demonstrator, using a different initial set of
|      beliefs.
*/

bw1 :-
        Beliefs = [clr(a),
                clr(b),
                    clr(c),
                  clr(t),
                  on(a,t),
                     on(b,t),
                     on(c,t),
                     block(a),
                     block(b),
                     block(c),
                     table(t)],
        survive(Beliefs,bwc).



/*
| bw2: another blocks-world demonstrator, using a different initial set of
|      beliefs.
*/

bw2 :-
        Beliefs = [clr(a),
                clr(b),
                  on(a,t),
```

```
                block(a),
                block(b)],
        survive(Beliefs,bwc).



/*
| rw: a room-world demonstrator.
*/

rw :-
        Beliefs = [door(d1,open),
                light(l1,on),
                  colour(desk,brown),
                  colour(printer,white),
                  colour(sky,purple)],
        survive(Beliefs,rwc).



/*
| dcs(bwc,O): O is the set of blocks-world domain constraints.  A gloss:
|       1) Blocks are clear or under something;
|       2) Blocks are on at most one other thing;
|       3) Blocks are under at most one other thing;
|       4) Things that are tables are not blocks, and things that are blocks
|          are not tables;
|       5) Things are never on each other.
*/

dcs(bwc,[[on(X1,Y1),clr(Y1),block(Y1)],
        [on(X2,Y2),on(X2,Z2),block(X2)],
        [on(X3,Y3),on(Z3,Y3),block(Y3)],
        [block(X4),table(X4)],
        [on(X5,Y5),on(Y5,X5)]]).



/*
| dcs(rwc,O): O is is the set of room-world domain constraints.  In English:
|       1) Lights are on or off;
|       2) Doors are open or closed;
|       3) Any given object has only one colour.
*/

dcs(rwc,[[light(L,on),light(L,off)],
         [door(D,open),door(D,closed)],
         [colour(O,C1),colour(O,C2)]]).
```

Appendix B:
Example Runs of Prolog Code.

```
Edinburgh Prolog, version 1.4  (6th October 1986)
AI Applications Institute, University of Edinburgh

| ?- ['pred-maint.pl'].

pred-maint.pl consulted:  11424 bytes      6.20 seconds

yes
| ?- bw.
 Current beliefs are:
  [clr(a),clr(t),on(a,b),on(b,t),block(b),block(a),table(t)]

 New observations? [block(c), on(c,t), clr(c)].
 Choices: [[]]
 Choosing to disbelieve: []
 Current beliefs are:
  [block(c),on(c,t),clr(c),clr(a),clr(t),on(a,b),on(b,t),block(b),block(a),
   table(t)]

 New observations? [on(c,a)].
 Choices: [[clr(a),block(c)],[clr(a),on(c,t)],[block(a),block(c)],[block(a),
         on(c,t)]]
 Choosing to disbelieve: [clr(a),block(c)]
 Current beliefs are:
  [on(c,a),on(c,t),clr(c),clr(t),on(a,b),on(b,t),block(b),block(a),table(t)]

 New observations? [clr(a)].
 Choices: [[on(c,a)],[block(a)]]
 Choosing to disbelieve: [on(c,a)]
 Current beliefs are:
  [clr(a),on(c,t),clr(c),clr(t),on(a,b),on(b,t),block(b),block(a),table(t)]

 New observations? [on(b,c)].
 Choices: [[on(b,t)],[block(b)]]
 Choosing to disbelieve: [on(b,t)]
 Current beliefs are:
  [on(b,c),clr(a),on(c,t),clr(c),clr(t),on(a,b),block(b),block(a),table(t)]

 New observations? q.
 Finished

yes
```

```
| ?- bw1.
 Current beliefs are:
  [clr(a),clr(b),clr(c),clr(t),on(a,t),on(b,t),on(c,t),block(a),block(b),
   block(c),table(t)]

 New observations? [block(t)].
 Choices: [[on(c,t),on(a,t),on(b,t),table(t)],[clr(t),on(c,t),on(a,t),
          table(t)],[clr(t),on(c,t),on(b,t),table(t)],[clr(t),on(a,t),
          on(b,t),table(t)],[clr(t),on(b,t),on(c,t),on(a,t),table(t)]]
 Choosing to disbelieve: [on(c,t),on(a,t),on(b,t),table(t)]
 Current beliefs are:
  [block(t),clr(a),clr(b),clr(c),clr(t),block(a),block(b),block(c)]

 New observations? q.
 Finished

yes
| ?- bw2.
 Current beliefs are:
  [clr(a),clr(b),on(a,t),block(a),block(b)]

 New observations? [on(b,t)].
 Choices: [[]]
 Choosing to disbelieve: []
 Current beliefs are:
  [on(b,t),clr(a),clr(b),on(a,t),block(a),block(b)]

 New observations? [table(b)].
 Choices: [[block(b)]]
 Choosing to disbelieve: [block(b)]
 Current beliefs are:
  [table(b),on(b,t),clr(a),clr(b),on(a,t),block(a)]

 New observations? [on(a,b)].
 Choices: [[on(a,t)],[block(a)]]
 Choosing to disbelieve: [on(a,t)]
 Current beliefs are:
  [on(a,b),table(b),on(b,t),clr(a),clr(b),block(a)]

 New observations? q.
 Finished

yes
```

```
| ?- rw.
 Current beliefs are:
  [door(d1,open),light(l1,on),colour(desk,brown),colour(printer,white),
   colour(sky,purple)]

 New observations? [door(d1,closed), light(l1,on)].
 Choices: [[door(d1,open)]]
 Choosing to disbelieve: [door(d1,open)]
 Current beliefs are:
  [door(d1,closed),light(l1,on),colour(desk,brown),colour(printer,white),
   colour(sky,purple)]

 New observations? [light(l1,off), colour(desk,green), colour(printer,blue)].
 Choices: [[light(l1,on),colour(desk,brown),colour(printer,white)]]
 Choosing to disbelieve: [light(l1,on),colour(desk,brown),
                          colour(printer,white)]
 Current beliefs are:
  [light(l1,off),colour(desk,green),colour(printer,blue),door(d1,closed),
   colour(sky,purple)]

 New observations? [colour(sky,yellow)].
 Choices: [[colour(sky,purple)]]
 Choosing to disbelieve: [colour(sky,purple)]
 Current beliefs are:
  [colour(sky,yellow),light(l1,off),colour(desk,green),colour(printer,blue),
   door(d1,closed)]

 New observations? q.
 Finished

yes
| ?- halt.

Prolog terminated
```

Appendix C:
The Order-Independence of a Detached Set of E-elements.

A simple proof of the order-independence of a detached set of e-elements.

If a step is finite, then it can be realised by the individual assumptive application of its component e-elements in arbitrary order.

Formal statement.

Let $P = (B,E;F,A)$ be a plan, let $X \subseteq \Pi(L_B)$ be a set of domain constraints, let c, $c' \subseteq L_B$ be cases, and let $G \subseteq E$ be a finite assumptive step from c to c', respecting X.

Let $(e_1, e_2, \cdots, e_n)$ be an arbitrary ordering of the elements of G such that $G = \{e_1, e_2, \cdots, e_n\}$. Then there are cases $c_0, c_1, \cdots, c_n$, such that $c = c_0$, $c' = c_n$, and $c_{i-1} \overset{X}{\underset{e_i}{\Rightarrow}} c_i$ $(i = 1, \cdots, n)$.

Proof.

Let $e_1, e_2, \in G$, and let c be a case in which both e_1 and e_2 have concession. Assume that $c \overset{X}{\underset{e_1}{\Rightarrow}} c'$. By the definition of detachment, e_2 has concession in c'. Assume instead that $c \overset{X}{\underset{e_2}{\Rightarrow}} c'$. Once again, by the definition of detachment, e_1 has concession in c'. For $i = 1, \cdots, n$ it follows that each e_i remains enabled during the successive occurrence of $e_1, \cdots, e_{i-1}$, and that under assumptive occurrence e_i can transform c_{i-1} into c_i. This works for an arbitrary ordering of the elements of G, so it therefore works for all orderings of the elements of G.